AD-784 824

ON A MEASURE OF PROGRAM STRUCTURE

Robert Noyes Chanon

Carnegie-Mellon University

Prepared for:

Defense Advanced Research Projects Agency

Air Force Office of Scientific Research

November 1973

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 74 - 1435	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 784 824
4. TITLE (and Subtitle) ON A MEASURE OF PROGRAM STRUCTURE		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert Noyes Chanon		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE November, 1973
		13. NUMBER OF PAGES 295
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research /NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Program structure has been discussed as being an important influence on the ease with which programs can be constructed, verified, understood, and changed. This thesis proposes a definition for program structure, a method for constructing programs, and a measure for program structure. The usefulness of this measure as a tool for determining and controlling structure is evaluated. The measure uses the information theoretic concept of excess entropy to determine the extent to which assumptions made by identified parts of programs are shared and thus influence structure. Several programs are developed using

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

I

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (abstract cont.)

mechanical aids to record assumptions and computer entropy loadings. These applications of the measure are evaluated and discussed.

ON A MEASURE OF PROGRAM STRUCTURE

Robert Noyes Chanon

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

November, 1973

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence [Contract F44620-73-C-0074] monitored by the Air Force Office of Scientific Research and by the National Science Foundation [Contracts GJ 37728 and GJ 32259]. This document has been approved for public release and contract sale; its distribution is unlimited.

ABSTRACT

Program structure has been discussed as being an important influence on the ease with which programs can be constructed, verified, understood, and changed. Yet the notion of program structure has remained a vague and imprecisely defined concept. This thesis proposes a definition and a measure for program structure and evaluates the usefulness of the measure as a tool for determining and controlling structure in a program.

Applications of the measure require that the assumptions which objects make be precisely stated. These are defined to include assumptions about the nature and use of variables and data; conditions relating to the correct execution of the program; and assumptions about the program environment in which the text is executed. Top-down programming by stepwise refinement forms the basis for a proposed methodology that permits these assumptions to be stated as a program is constructed.

The measure uses the information theoretic concept of excess entropy - entropy loading - to determine the extent to which assumptions are shared. Entropy loading calculations also provide a way of comparing different decompositions of a program. Unfortunately, finding the best decompositions of all but small programs seems intractable. Consequently, several heuristics are stated that attempt to establish bounds on the growth of entropy loadings for elaborations of decompositions suggested at early stages in a development.

Several programs are developed using mechanical aids to record assumptions and compute entropy loadings. Since each development preserves assumptions at every elaboration, this information need not be deduced from program text when the program is studied or is to be modified. Entropy loading figures at each stage allow different decompositions to be compared and provide either a basis for choosing a decomposition or grounds for actually modifying the program to achieve better structure. These developments illustrate the proposed methodology and show that the measure produces results that are usually consistent with the definition of program structure as well as the informal notion of structure from the literature.

Without mechanical aids, however, applications of these techniques to practical problems would be tedious and difficult. This and other difficulties motivate further research about this important but elusive property of programs: their structure.

ACKNOWLEDGEMENTS

I would like to express my appreciation to David L. Parnas for his patience and guidance during the early stages of research for this thesis. Special thanks are also due to the members of the thesis committee for their persistent criticism and helpful suggestions: John McCredie, Mary Shaw, William Wulf, and John Grason.

This thesis has relied heavily on the work of E. W. Dijkstra as a source for examples and intuitive notions about the meaning of structure in programs.

TABLE OF CONTENTS

INTRODUCTION

Dijkstra[DJ1-5], Wirth[W], Naur[NA1-2], Parnas[PA1], and Mills[MI1-2] have stressed the importance of designing software as collections of small programs, whose interrelations are well understood. Such software is said to have good structure. Unfortunately, not every piece of software which consists of a collection of small programs has good structure. Nor do the informal methods, described in the papers above, necessarily guarantee good structure. The goal of this thesis has been to investigate the behavior of a mathematical tool - entropy loading - as a measure of the goodness of structure and as a guide which can help to preserve good structure in a collection of programs that constitutes the decomposition of a piece of software.

The work of Simon[SI] and Alexander[AL], concerning complexity in systems, and the work of Dijkstra[DJ1-5] and Parnas[PA1] strongly suggest that system decompositions having good structure possess the following properties:

(1) the information required to study, understand, and verify single parts of a system is supplied in conjunction with those parts, and relatively little information about the rest of the system is required;

(2) single parts can be drastically changed - changed algorithm, changed data structures - without requiring much knowledge of the rest of the system and without changing the rest of the system, i.e. drastic changes can actually be confined to single parts;

(3) should an error occur as a result of the failure of one small part to function correctly, the error can be localized to that part of the system quickly and easily, permiting the error to be repaired using only a knowledge of that part;

(4) during system construction, distinct working groups can be given assignments to write separate sets of parts, and the assignments can be completed with very little communication among the groups.

Many programs, even some that are regarded as good programs, fail to possess one or several of these properties - with undesirable results. Programs that fail to meet these standards frequently require that many design decisions be understood before a small portion of the program can be understood. Seemingly unrelated portions often make many subtle assumptions about each other, assumptions that are difficult to deduce from the program text or documentation. Hence, in order to change a portion, the consequences of the change must be understood in the context of a large part of the program. Shared assumptions between identified parts will be called interactions. Simon[SI] and Alexander[AL] suggest that by controlling these interactions, limiting the amount of code that each connects, and displaying the structure of the program, the four properties mentioned above are approached.

In order to control interactions, it is first necessary to explicitly observe the assumptions made by parts of a program. Techniques for making these observations are demonstrated, along with a tabular format for recording them once they have been noted. This record can then be used to compare different decompositions of the program. The comparisons are made by using an entropy loading calculation described by van Emden[vE2]. The calculation has long been used in areas such as information theory, physics, and ecology. This thesis investigates its applicability to data representing the assumptions of program parts. The entropy loading measure was chosen because it had the potential for distinguishing, in a mathematical way, programs that possess properties consistent with good structure from programs that do not. This mathematical tool enables a designer to note the existence of interactions between the parts of his design. As a result, he should be able to better understand and to control the effects of his assumptions.

Applications of these methods produce results that are usually consistent with intuitive notions - in the sense of Dijkstra and Wirth - about what constitutes good program structure. Programs that have been regarded as "good" are shown to have better properties than programs regarded as "bad". Several "good" programs have even been improved. Several anomolies of the measure as well as the costs and difficulties encountered while applying it are also discussed.

The chapters that follow give a definition of structure and interaction and show how the effects of interactions can be measured. These techniques are also demonstrated on numerous examples that have been used elsewhere [DJ3,W,HE,MK1].

Specifically,

Chapter I examines part of the literature about software design. It then presents a definition of program structure that motivates the techniques used in this thesis.

Chapter II describes how a design methodology - structured programming[DJ1] - and a proof technique[HO1] can be used to find the interrelations in a program. Interrelations are observed in an example and summarized in object/assumption tables.

Chapter III presents an entropy loading measure that can be used to formally evaluate the relative merits of different decompositions of a program. The advantages as well as the shortcomings of the measure are discussed.

Chapter IV demonstrates the use of the measure to observe and control program structure. Several examples are developed using the heuristics described in Chapter III.

Chapter V discusses and evaluates the results of this thesis.

Finally, two appendices follow the conclusions: the first presents a larger example and the second uses the measure as a basis for discussing the paper, **Compiler Structure**[MK1].

CHAPTER I

REVIEW OF PREVIOUS WORK

This chapter begins by stating several properties of programs which seem to have good structure. Next, several design methodologies and conventions for representing programs are examined to see to what extent they lead to good structure. Each is shown to have the potential for leading to bad structure unless the issue of structure is explicitly considered. As a result of this examination, a definition of program structure is proposed in terms of the interrelations among the parts of a program. The analysis and control of structure motivates the methodology used in this thesis.

ASPECTS OF GOOD STRUCTURE

Dijkstra("On Our Inability To Do Much"[DJ1,pp 1-3]) points out that a person's power of comprehension is too meager to deal with all the detail in a large program. However, he asserts that the computer owes its existence to its ability to execute large, complicated programs. Consequently, we must find ways to organize large programs which allow people to deal with them, yet utilize a computer well. Several properties of such programs are:

(1) Ease of verification

> Good programs can be proved correct by dealing with only small parts of the program, regardless of whether the proof is carried out by a person or a machine.

(2) Ease of understanding

The text of a good program, along with its documentation, describes the program in sufficient detail that it can be understood, and

(a) Single parts of good programs are understandable in terms of their text and documentation. (In fact, the documentation of a good program is not separate from the program.)

(b) There is an understandable path from the abstract method being implemented to the final detailed code.

(c) Decisions along this path are explicit and can therefore be evaluated as either good or bad.

(3) Ease of maintenance and change

Good programs, because the effects of single parts and the assumptions made by them are understandable, can be changed and maintained with relative ease. Little effort need be spent deducing the consequences of changes when compared with the effort needed to implement the changes.

Dijkstra has demonstrated the role the structure of a program plays. His observations were motivated by the verification issue. For example, he cites the problem of verifying the correct behavior of a hardware multiplier[DJ1]. If we regard the device as a "black box", then all possible multiplications must be performed and verified as being correct. Alternatively, the internal structure of the device can be examined, and a convincing argument about the correctness of the device can be produced. This second approach is the only feasible one.

The structure of a program has direct effects on the three properties of good programs mentioned above. Some of these effects are:

(1) Program verification is tractable if the amount of detail required for the verification can be comprehended by the human or mechanical verifier. This means that the parts of a program and the relations among those parts are sufficiently simple that theorems relevant to the correctness question can be easily stated and easily proved.

(2) "Understanding a program" is a less formal view of program verification. When we say that we "understand" a program we mean that we understand how it works - that we believe it behaves the way it is purported to behave. Since we don't attempt to understand programs as "black boxes", the single parts and their interrelations must be sufficiently simple that understanding is possible.

(3) Maintaining or changing a program requires that the places to change be found, the changes constructed, and the constructions verified. This implies that the parts to be changed must not only be identifiable to a programmer but that the effects of those changes in the context of the entire program be limited. Hence, the parts which are to be changed and their interrelations with the rest of the program must be kept within a programmer's grasp.

INFLUENCING STRUCTURE - A REVIEW OF THE LITERATURE

A few techniques have been proposed to help control program structure.

Knuth[KN1] has described a "classic" paradigm in which a system is designed from the "top down" in terms of subroutines, and coded from the "bottom up". He emphasizes the mechanism of the subroutine as a tool for allowing attention to be focused on one design decision, postponing the consideration of other details. This approach can help to display the structure of a program. However, the desirable properties of the programs which the methodology produces are never stated.

Wirth[W] provides examples where program text is written in a top-down, step-wise manner. His model pays particular attention to the importance of verification and mentions the potential difficulties which might accompany an attempt to change a program. The Eight Queens

Problem, which is presented as an example in Chapter IV, and based on the development by Wirth, shows a situation where decisions about control flow have been separated from decisions about data representation.

Naur[NA2] states a specific criterion which he claims should be used to guide decision making in the design process. He proposes that the global requirements of a problem should be used to deduce a set of "associated actions which guarantee that these requirements are always met" - **programming by action clusters.** Here the importance of describing the effects of these operations precisely is stressed. Program verification then becomes more tractable than otherwise. Unfortunately, the initial statement of the problem used to demonstrate the method is imprecise, and design decisions which precisely define the problem to be solved are never stated but simply appear in the code. Further, one of the global requirements must be considered in at least one place other than the action cluster which was specifically associated with it. This oversight regarding interrelations among the program parts results in an error which is documented by Leavenworth[LE]. In addition, Naur emphasized the importance of considering the relationships of global properties as a program is written, but made no attempt to display these relationships.

The issue of structure was first stressed by Dijkstra[DJ1] in terms of a programming methodology called "structured programming". Dijkstra has carefully chosen his examples. Each represents the outcome of a careful search in which many choices were rejected. These choices and the reasons for their rejection are seldom stated.

Each example problem is presented in terms of a collection of carefully derived sub-problems which together solve the original problem. The arrangement and relationships of these sub-problems allow the correctness of his programs to be verified as they are written. Verifications are established by informal proofs of correctness and by direct applications of an axiom system due to Hoare[HO1]. Dijkstra has shown that verification in terms of the representations suggested by the programming methodology is tractable for systems which are not just examples of the methods[DJ4].

Lastly, Dijkstra recognizes the importance of producing programs which are maintainable and changable ("On Program Families"[DJ1]).

Mills[MI1,MI2] and Baker[BA] have adopted the structured programming methodology to the extent that precise coding conventions are selected in order to convert programs containing parts which have yet to be elaborated directly into code. Their conventions are justified by a "structure theorem" due to Boehm and Jacopini[as described in an appendix to MI1]. These conventions include maintaining a program library of (perhaps) dummy entries so that high level program texts - containing references to code which has yet to be elaborated - can be compiled and verified. They also observe a convention where PL/I text representing any single program part is always less than a page (55 lines) in length. The verification aspect receives specific attention through attempts to formally verify texts. Besides formal verification, several programmers read the texts and either conclude that they are correct or correct them. Some of these conventions, however, can lead to bad structure. For example, if programmers are encouraged to read the texts of other programmers, it is possible that unstated assumptions which would adversely affect the maintainability and changability of the program will be made. Much attention is paid to a variety of control structures eg., texts must be **go to**-less.

Besides these methodologies, efforts have been made to design and implement programming languages which restrict the kinds of syntactic structures that can actually be used to represent programs. The intent is to eliminate the use of language constructs which have a high probability of leading to poorly structured, complex code. For example, without adequate descriptions of the relationships among various pieces of code in a program, the unbridled use of the **go to** statement makes it impossible to guarantee good properties in all but the smallest programs. Dijkstra[DJ6] noted this difficulty and suggested that the **go to** statement be avoided. As a result, at least one programming language (BLISS[WRH]) has no **go to** statement and **go to**-less programming is advocated as a good programming practice. Indeed, some **go to**-less programs do not possess many of the bad properties of some programs containing **go to** statements, but such programs, just by virtue of the absence of the **go to**, are not guaranteed to have good structure.

Some so-called implementation languages[W1,WRH] have attempted to make it possible for cooperating programmers to write good programs. These languages have eliminated or restricted the use of many syntactic constructs which seem to lead to bad structure. None, however, is accompanied by a methodology which suggests how it should be used to produce programs which have good structure. That this is a severe shortcoming is apparent in the light of the developments of TSS/360 and the MULTICS operating systems: TSS was written in the assembly language for the IBM 360/67, MULTICS in an implementation language based on PL/I. Both systems have similar goals. However, MULTICS encountered many of the same development problems as did TSS.

Snowdon[SN] has described an interactive system which attempts to provide an environment which allows programs to be expressed in much the same way as they would be developed using the structured programming methodology. But, even here, there is no specific description of how the language is meant to be used. Further, no emphasis is placed on the manner in which different "abstract machines" are or should be related to one another. Using the language in no way guarantees good structure in the resulting programs.

Parnas[PA1-PA4] has addressed the issue of how to produce pieces of software which have good properties by stressing the importance of precise specifications for the independent modules which make up the piece of software. The policy of "hiding" information which a module does not need is used extensively. The only information which is available to the implementor of a module (and hence to the module) is presented in the specifications of the modules. The effect of such a policy is to restrict interrelations among modules to those which can be deduced from the specifications. (These may still be very subtle.) Systems which are produced using this methodology have the following properties:

(1) Once the specifications for each module have been written, the system can be constructed in a straightforward way, based solely on the information contained within the specifications.

(2) The system can be verified as correct if the specifications lead to a correct solution of the problem for which the system was designed and if each module can be verified as meeting its specification.

(3) The overall system is understandable if its behavior can be deduced from the specifications with reasonable ease. Each module is understandable if:

(a) the interrelations among the functions which comprise it are understandable and

(b) the implementation of each function is understandable.

(4) Individual portions of the system can be changed so that the good properties of the system are preserved, but only if those changes are made within single modules. So long as a module meets its specifications, it can be freely changed. Extensions to the system, however, imply changes to the specification itself. Such changes must be consistent with those portions of the system which remain[PA4].

If the interrelations which are deducible from the specification are numerous and complicated, a system may have bad properties (Note the parallel between this phenomenon and programs produced using the structured programming methodology). Assigning tasks to modules is as important as the specifications, which may, themselves, be difficult to write. Parnas' work has been criticized because he provides incomplete guidelines about how module partitions should be constructed. However, the work provides a framework which has good implications for ease of understanding and ease of maintenance and change. It also provides a model which has been successfully used in constructing software in many versions with different implementations of each module.

PROGRAM STRUCTURE: A DEFINITION

Dijkstra[DJ1-DJ4] has demonstrated a process by which he constructs programs that have good structure but has not emphasized the properties of the relationships among the parts which make the structure good. In all his examples, the relationships among the program parts have been few in number and so straightforward as to be easily neglected. In every instance, the relationships among the sub-parts inside a part have been far more numerous than the relationships among the parts. Parnas[PA1] stated a definition of program structure in terms of **modules** and **connections**. The modules are those portions of a program which are specifically indicated in the written description of the program - perhaps its documentation. The connections among the modules "**are the assumptions which the modules make about each other.**" These connections are much more extensive than the calling sequences and control block formats shown in most descriptions. The definition stated below is a modification of the definition due to Parnas. Objects are construed to be any program parts which have net effects on the state of the program or its data. The term **interaction** is further defined as a shared assumption among two or more objects. Hence, the **assumptions** which the objects of a program make include the **connections**, which in turn include the **interactions**. Consequently, we have the definition:

> **Program Structure** is the set of interactions which exist among identified objects in a program as well as the ways those objects are organized to form the whole program.

The definition implies that any program possesses structure. The aspects of good structure at the beginning of this chapter, however, imply that programs having good structure are constructed from objects whose interactions and assumptions are apparent or are easily deducible. The key emphasis in this definition is that good structure is determined by interactions among objects and not just the organization of the objects.

The definition motivates the model which is used in this thesis to develop and represent programs. This model is developed in Chapter II, but has two properties which arise directly from the definition of program structure.

(1) Objects are constructed in an organized way, using the top-down and step-wise construction techniques of Dijkstra and Wirth.

(2) Interactions are explicitly recognized and recorded and are used to suggest ways of maintaining good structure in a program.

SUMMARY

This chapter first presented a list of several aspects of good program structure. Next, several programming methods were discussed with respect to the ways they influence program structure. Lastly, a definition of program structure, emphasizing interactions among the objects of a program, was stated. This definition motivates the methodology that is described in Chapters II, III, and IV.

In Chapter II, the nature of these interactions is investigated and a proposal for keeping track of them is made.

CHAPTER II

MAKING ASSUMPTIONS EXPLICIT

This chapter first describes the kinds of assumptions which have been observed among the objects of a program and then describes a methodology for actually displaying them. Next, object/assumption tables are demonstrated as a way of recording observed assumptions. In chapter III, such tables will be used to state the measure of program structure which is investigated in this thesis. In the remaining chapters, object/assumption tables will be maintained for each example to which the measure is applied. Finally, an example program is examined in order to observe the assumptions made by its objects and to demonstrate the proposed methodology.

OBJECTS AND ASSUMPTIONS

Chapter I asserted that program structure is determined by the objects of a program and their interactions, where interactions are defined to be assumptions shared among objects. In point of fact, objects don't make assumptions. Rather, assumptions are made by a designer/programmer and are used as guides to construct objects. Dijkstra[DJ1,DJ2] has displayed objects in the form of English statements which describe the intent of various parts of, as yet, incomplete programs. Seldom are the assumptions which were used to construct them stated precisely. This situation first poses the question of what kinds of assumptions are made.

The classification of assumptions below has sufficed for the examples developed in this thesis. Since the notion of an assumption is subjective, other researchers might wish to add to this list.

(1) Relationships which must hold prior to the execution of an object in order for its effects to be realized.

(2) Assumptions about data, e.g. assumptions about the meaning and interpretation of values contained in simple variables or data structures; assumptions about the position of information in data structures; assumptions about accessibility of data; etc.

(3) Assumptions about the environment in which an object is executed, e.g. frequency of use of an object; order in which computations will be performed; machine precision; assumptions about factors outside the control of the program.

(4) Assumptions based upon mathematical theorems which are relevant to the problem being solved.

(These classifications will be referred to in the next section.)

Next, the problem arises as to how these assumptions can be found and stated. A tempting approach would be to examine a complete program, understand it, and record the assumptions which objects make - the objects being deduced by the examiner from the code and its documentation. In all but the smallest programs, this approach is extremely difficult. Both objects and their assumptions must be deduced from detailed code. This is a result of the so-called abstraction/implementation dilemma, i.e. it is frequently possible to find a program which implements an abstraction, but it is usually difficult to deduce the abstraction from an implementation of that abstraction.

The only feasible approach is to observe assumptions as **a program is being constructed.**

A DESIGN METHODOLOGY FOR PRESERVING ASSUMPTIONS

The proposed methodology is based upon the "top-down", "step-wise refinement" methods suggested by Dijkstra and Wirth. We begin by describing the original problem in terms of a set of assumptions that constrain the problem and a set of final conditions (post-conditions) that describe the net effects the program is intended to produce. Here, it is vital that the specifications of the problem be precisely stated. A notation due to Hoare[HO1, HO2] will be used to describe this situation:

$$P \{S\} Q$$

Here, P are the assumptions, Q are the post-conditions, and S is program text or an explicit description of what an object does. Next, S is described as an arrangement of "simpler" computations whose effects are meant to lead from P to Q. Each "simpler" computation possesses its own pre-conditions and post-conditions. Clearly, this descriptive process can be continued as deeply as necessary. We identify these "simpler" computations - as well as the original S - as the objects referred to in the definition of program structure. In the past, objects have been described informally in English. Such descriptions serve only as reminders for what the assumptions and post-conditions of an object really are. This kind of informality can allow important detail to be neglected and can obscure the intent of the object. Henderson and Snowdon[HE] showed an example where this difficulty actually led to an incorrect program.

The assumptions, post-conditions, and intended effects of each object must be precisely described so that a program can be verified. In the example which appears at the end of this chapter, such a correctness argument will be presented explicitly.

The assumptions which are used to construct an object will be identified by first noting the post-condition which is meant to hold after the object is executed. Next, the effects of the object will be examined in order to answer the following four questions:

(1) What relationships "must" hold prior to the execution of the object in order for the effects to lead to the post-condition? (The answers to this question are the "weakest" such relationships.)

(2) What kinds of actions are needed, permitted, or used with respect to the data or variables mentioned in the object to insure that the post-condition holds?

(3) What information about the context in which the object is meant to execute is needed, permitted, or used to insure the post-condition?

(4) What theorems are needed or used by the object and what theorems is the object permitted to use in order to insure the post-condition?

Each question corresponds directly to one of the four classifications of assumptions listed at the beginning of this chapter. The answers to these questions constitute a set of distinct assumptions which are meant to include the **connections** and **interactions** described in Chapter I. A record of these assumptions will be used by the measure which is described in Chapter III. These questions are meant to be guides for finding those relationships or requirements which seem to determine program structure. As the examples will indicate, these kinds of assumptions, when associated with objects, allow applications of the measure to yield values which are usually consistent with the definition of structure from Chapter I.

The answers to question (1) have been characterized, in part, by Dijkstra[DJ5, yet unpublished], as the **weakest pre-conditions** of an object. Dijkstra has described rules for finding these weakest pre-conditions for assignments, conditional statements, and **while** statements, as well as certain kinds of recursive procedures. These rules, called **predicate transformers**, will be used to help derive assumptions for objects in the examples. It should be noted, however, that these predicate transformers are specific to the target language into which the examples are developed. (The following paragraphs define **weakest pre-conditions** and **predicate transformers** in more detail. These paragraphs may be skipped during a first reading of this thesis.)

* * *

Specifically, if P represents a post-condition for an object S and fS is the predicate transformer for S then fS(P) represents the weakest precondition for S which guarantees that P will hold after an execution of S. Dijkstra provides criteria for finding predicate transformers that can be applied to other kinds of syntactic constructs as well. The fundamental idea is to find consistent rules that provide weakest pre-conditions, given a post-condition and a particular instance of some construct. Even if such a rule cannot be found in general, an assumption can usually be constructed for specific cases that are of interest. For example, if S is an assignment, i.e. $X \leftarrow E$, then

$$f\text{"ASSIGNMENT"}(P) = P: E \rightarrow X$$

where E is the expression being assigned; X is the variable which is assigned the value of E; and $P: E \rightarrow X$ is the predicate which results by replacing all occurrences of X in P by E. For example, if S is $d := c + d$ and P is $a * c + b * d = A * B$ then the weakest pre-condition which results from applying the predicate transformer is $a * c + b * (c + d) = A * B$.

The predicate transformer for binary selection

(if B **then** S1 **else** S2)

is

$$f\text{"BINARY SELECTION"}(P) = (\ B \wedge fS1(P)\ ) \text{ or } (\ \neg B \wedge fS2(P)\ )$$

Similarly,

$$f\text{"CONCATENATION"}(P) = fS1(\ fS2(P)\ )$$

is the predicate transformer for S1 ; S2.

Besides displaying several predicate transformers, Dijkstra has proven a theorem about predicate transformers which is based on the following definition:

> If two predicate transformers fS and fS' satisfy the property that for all P, $fS(P) \supset fS'(P)$ then "fS is **as strong as** fS'" and "fS' is **as weak as** fS".

The theorem can now be stated:

> **Theorem of Monotonicity:** Whenever in a predicate transformer fS, formed by concatenation and/or selection and/or recursion, one of the constituent predicate transformers is replaced by one as weak(strong) as the original one, the resulting predicate transformer is as weak(strong) as fS.

For constructions such as **while** B **do** S1 or **repeat** S1 **until** B explicit predicate transformers are not presented. Instead, for the **while** construction, the Fundamental Invariance Theorem for Repetition has been proven, i.e.

If S is **while** B **do** S1 and Q is a post-condition for S1, then

$$((Q \wedge B) \supset fS1(Q)) \supset ((Q \wedge fS(\mathbf{true})) \supset fS(Q \wedge \neg B)).$$

This theorem along with the Theorem of Monotonicity can be used to find pre-conditions which imply the weakest pre-condition for a **while** construction or a **repeat** construction. For example, if

$$((Q \wedge B) \supset fS1(Q)) \wedge (Q \wedge fS(\mathbf{true}))$$

is used to replace the weakest pre-condition for a **while** construction, given $(Q \wedge \neg B)$ as the post-condition, then the Theorem of Monotonicity asserts that the pre-conditions which result from applications of selection, concatenation, and recursion will be as strong as any formed by using the weakest pre-condition for the **while** construction. Further, if

S: **repeat** S1 **until** B

is interpreted as

S1; S': **while** ¬B **do** S1;

then

$$fS1(((Q \wedge \neg B) \supset fS1(Q)) \wedge (Q \wedge fS'(\mathbf{true})))$$

is as strong as $fS(Q \wedge B)$. A difficulty arises, however, when we attempt to display $fS(\textbf{true})$ whenever S is a **while** construction. Informally, $fS(\textbf{true})$ can be interpreted as the "weakest pre-condition for S which guarantees that S terminates". It is frequently possible to find conditions which guarantee termination, but it is more difficult to prove that a particular termination condition is the weakest. However, any condition for termination, say X, implies the weakest pre-condition for S to terminate, i.e. $X \supset fS(\textbf{true})$ so that the Fundamental Invariance Theorem for Repetition allows

$$((Q \wedge B) \supset fS1(Q)) \supset ((Q \wedge X) \supset fS(Q \wedge \neg B))$$

Hence, if $((Q \wedge B) \supset fS1(Q)) \wedge (Q \wedge X)$ is used to replace the weakest pre-condition for a **while** construction, given $(Q \wedge \neg B)$ as the post-condition, then, subject to the conditons of the Theorem of Monotonicity, this predicate will be as strong as any formed by using the weakest pre-conditon for the **while** construction.

* * *

The notion of a "weakest pre-condition" can be used informally to suggest essential relationships which must hold prior to the execution of an object which is not yet represented in the target language. In short, these assumptions should insure that the object is correct, given the post-conditions which hold after the object is executed.

"Weakest pre-conditions" are simply relationships which must hold prior to the execution of an object and do not include descriptions of the capabilities which must be available to the objects. For example, in order to execute an assignment, access rights for variables are not mentioned explicitly, i.e. if a post-condition for $d := d + c$ is $a * c + d * b = A * B$, then even though the assumption $(a + b) * c + b * d = A * B$ must hold, the assignment requires write access to d and read access to d and c, but requires no access to a, b, A, or B. Similar arguments can be made for examples of assumptions about program environment and assumptions which are theorems from mathematics, i.e. assumptions which are answers to questions (3) and (4).

Assumptions associated with objects that occur at early stages of a development must usually be deduced informally. At later stages, formal methods - such as the techniques of Dijkstra[DJ5] - can be used to find some of the assumptions. It should be clear, however, that the depth to which refinements are made is up to the discretion of the designer. In some of the examples, refinements have been made to the statement level in order to apply Dijkstra's techniques directly. In general, refinements which imply greater detail than the constructs for which a set of predicate transformers have been found seems ill-advised since that kind of detail involves operations which are available to all the objects. The section in Chapter III which discusses the probability of change of assumptions provides further evidence to justify this choice.

Since interactions are of major interest when structure in a program is examined, the assumptions of an object should be represented as a conjunction of single assumptions. This kind of representation allows interactions to be observed easily. In particular, the next section describes a tabular scheme for recording assumptions where this kind of representation is particularly helpful.

To summarize, the assumptions associated with an object should describe its requirements as completely as possible.

The question now arises as to how objects should be elaborated. Each object will be elaborated as an arrangement of sub-objects, each possessing its own assumptions and post-conditions. Frequently, such an elaboration may not require all the assumptions which are made by the object which is being elaborated. For example, if

$$P \{S\} Q$$

where P represents the assumptions of an object {S} with post-condition Q and X and Y can be found such that $P \supset X$ and $Y \supset Q$ and $X \{S\} Y$ then the elaboration of {S} can be made using only the assumptions implied by X, rather than all those implied by P.

As an aid to finding X and Y, we adopt the guideline suggested by Parnas[PA1] where we attempt to describe {S} via a specification which "hides" as many of the assumptions in P as possible. This same guideline can be applied at the initial stage of program development as well. Any time this guideline is used, however, we include $P \supset X$ and $Y \supset Q$ as assumptions of S.

A common example of information hiding, where these additional assumptions will **not** be included, involves concatenations of several objects, perhaps the result of some elaboration. Weakest pre-conditons associated with a concatenation of several objects frequently include clauses which are of no relevance to individual objects in the concatenation. For example, if a post-condition for $x := x + 1; y := y - 1;$ is $x > 4 \wedge y < 2$ then the weakest pre-condition for $x := x + 1$ is $x > 3 \wedge y < 3$. In order to satisfy the post-condition for the concatenation, the first assignment required no information about x and the second required no information about y, but their combined effects led to the post-condition. For this reason, in a concatenation only those clauses in a post-condition which are changed as a result of the effects of an object will be listed as assumptions for that object. This choice can be justified by observing that if $A \wedge B \{S\} C \wedge B$ and no elaboration of S can be made such that $A \{S\} C$ makes B **false** then $A \{S\} C \wedge B$. Since this is just a theorem from mathematics, the section which discusses **Probability of Change of Assumptions** justifies the omission of these unaffected clauses.

To summarize:

(1) The methodology addresses programs in a "top-down" fashion.

(2) The assumptions which are associated with each object are answers to the four questions stated earlier.

(3) If an object is elaborated and it is possible to find X and Y such that if $P\{S\}Q$ and $P \supset X$ and $Y \supset Q$ and $X\{S\}Y$ then $P \supset X$ and $Y \supset Q$ will augment the assumptions associated with S.

(4) Each program will be verified by showing that the assumptions associated with that object imply a pre-condition which was derived

by considering the effects of the object along with the post-condition which is to hold after it is executed.

OBJECT/ASSUMPTION TABLES

In order to record the assumptions which will be associated with objects, object/assumption tables will be constructed for programs. These tables will be used extensively in Chapter IV. The table is defined for all objects I and assumptions J such that

$$T(I,J) = \begin{cases} 1, & \text{if object I makes assumption J} \\ 0, & \text{otherwise} \end{cases}$$

In the examples, some care has been taken to represent assumptions as conjunctions of "simpler" assumptions, where each conjunct corresponds to a column in the object/assumption table.

AN EXAMPLE OF OBJECTS AND THEIR ASSUMPTIONS

The example which follows is meant to demonstrate the way assumptions are deduced and recorded, and should not be construed as typical of the amount of detail that should be preserved in all programs. The choice was made to refine objects so that the final program consisted of single statements or parts of statements, thus displaying explicit pre-conditions - in the Dijkstra sense - as well as assumptions of other kinds. As a consequence of this detail, the development proceeds very slowly if it is read from beginning to end. Nevertheless, if objects are to be understandable without additional context, assumptions must be stated wherever they are made. For this reason, a map is provided that summarizes the example. The example can best be studied by studying first the map and then the objects in terms of their assumptions, post-conditions, and effects. Postpone studying the verifications and object/assumption tables until the program itself is generally understood.

The objects of each of the programs in this thesis are described by:

(1) assumptions for the object.

(2) an explicit statement of the effects of the object, i.e. what this object actually does. (Note that the effects of an object are not necessarily the same as the post-conditions for an object, e.g. the post-conditions associated with a particular assignment statement may be quite different from the net effects of that assignment.)

(3) post-conditions for the object - a description of what every version of the object must do.

(4) a verification and check that the assumptions stated in (1) and the effects stated in (2) imply the post-conditions. (These verifications are informal. They are presented only for the first few examples.)

(5) a display of that portion of an object/assumption table which is appropriate to the object (assumptions which are added as a result of hiding information from an elaboration are indicated by a "+" in each table rather than a "1").

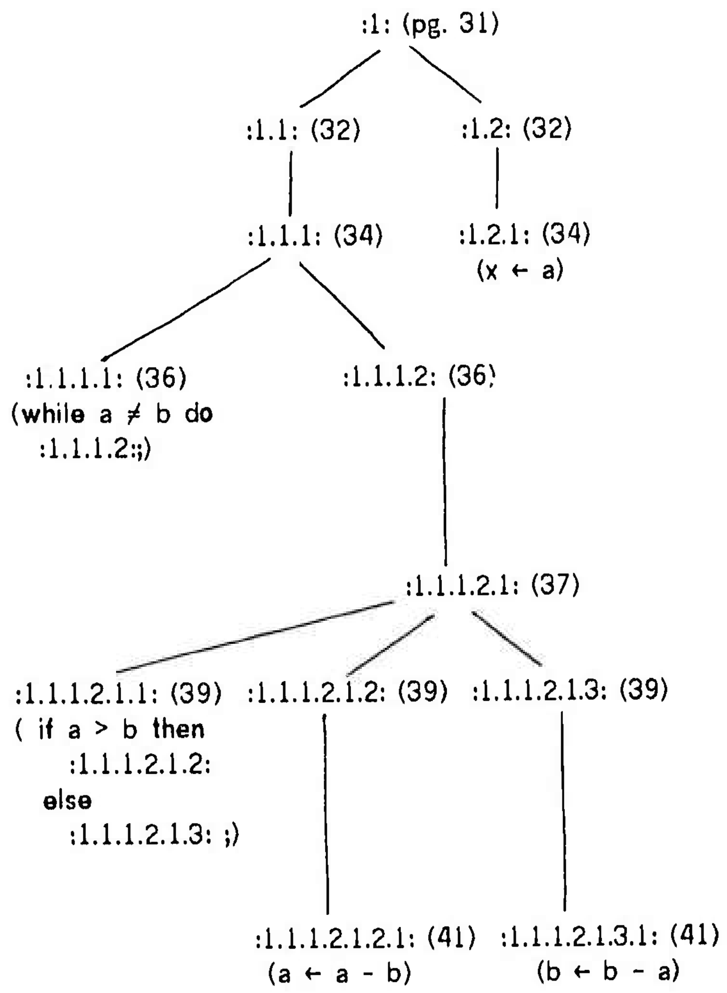
Each object will be named by an object name defined by:

```
<object name> ::= : <object number> :
<object number> ::= <positive integer> |
                    <object number> . <positive integer>
```

These names allow the ancestry of an object to be related to the rest of a program. For example, if object :10.5.3.2: is to be elaborated as three new objects, then the names for the sub-objects will be :10.5.3.2.1:, :10.5.3.2.2:, and :10.5.3.2.3:.

For the purposes of these examples, the target language for the programs can be regarded as a dialect of ALGOL 60 and is identical with a language used by Dijkstra[DJ3].

A GCD COMPUTATION



Object :1: assumes that positive integer values are contained in variables a and b and has the effect of assigning to x the value of the greatest common divisor of the initial contents of a and b (symbolized by A and B, respectively).

Object :1.1: leaves the value of gcd(a,b) in a and object :1.2: assigns the value of a to x.

The theorem that $(\gcd(a,b) = \gcd(A,B) \wedge a = b) \supset a = \gcd(A,B)$ is hidden from :1.1.1: and $a = \gcd(A,B)$ is hidden from :1.2.1:.

Object :1.1.1.1: controls the execution of :1.1.1.2: which maintains the invariant $(a < a' \vee b < b') \wedge \gcd(a,b) = \gcd(A,B)$, where a' and b' are the values of a and b prior to each iteration.

Object :1.1.1.2.1: hides the invariant $\gcd(a,b) = \gcd(A,B)$.

Finally, objects :1.1.1.2.1.1:, :1.1.1.2.1.2:, and :1.1.1.2.1.3: implement a conditional statement which results in $(a < a' \vee b < b') \wedge \gcd(a,b) = \gcd(a',b')$.

A GCD COMPUTATION

This example, based on a program due to Dijkstra[DJ3 pp. 33-41], is constrained so that it uses neither multiplication nor division to compute the greatest common divisor of two positive integers.

Compute the value of the greatest common divisor of the initial(positive) contents of the integer variables a and b, and leave the result in the variable x, without using either multiplication or division. The assumption that neither multiplication nor division is to be used in the program is an assumption which must be made by all the objects. It is identified as assumption "0" and in Chapter III will be shown to have no effect on the possible ways of decomposing the program.

object :1:

assumptions: a > 0, b > 0, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, write access to x is required, read and write access is required for both a and b, neither multiplication nor division is to be used

:1: COMPUTE THE GCD OF a AND b, LEAVE THE RESULT IN THE VARIABLE x, WITHOUT USING EITHER MULTIPLICATION OR DIVISION.

effects and **post-condition:** x = gcd(A,B)

verification: Since the gcd of any two positive integers is always defined and is computable, the computation is feasible.

The object/assumption table for :1: is

objects	0	1	2	3	4	5	6	7	8	9	10	11
:1:	1	1	1	1	1	1					1	1

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required

We elaborate :1: as

:1.1:

assumptions: a > 0, b > 0, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read and write access to a and b is required, neither multiplication nor division is to be used

:1.1: REPLACE THE VALUE OF a BY gcd(A,B)

effects and **post-condition:** a = gcd(A,B)

:1.2:

assumptions: a = gcd(A,B), write access to x is required, read access to a is required, neither multiplication nor division is to be used

:1.2: REPLACE THE VALUE OF x BY THE VALUE CONTAINED IN a.

effects: x = a

post-condition: x = gcd(A,B)

verification: In order for x = gcd(A,B) to hold after :1.2:, given that its effect is x = a, its assumptions must be a = gcd(A,B). But this is guaranteed as the post-condition of :1.1:.

object/assumption table for :1.1:, :1.2:

	0	1	2	3	4	5	6	7	8	9	10	11
:1.1:	1	1	1		1	1					1	1
:1.2:	1			1		1	1					

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) $a = \gcd(A,B)$
10) write access to b is required
11) read access to b is required

Consider next an elaboration of :1.1:. If the values of a and b can be modified so that $a = b$ and $gcd(a,b) = gcd(A,B)$ then $a = gcd(A,B)$. This means that by adding

$$(a = b \wedge gcd(a,b) = gcd(A,B)) \supset a = gcd(A,B)$$

to the assumptions of :1.1: we can write

:1.1.1:

assumptions: $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read and write access is required for both a and b, and neither multiplication nor division is to be used

:1.1.1: MAKE a = b SUCH THAT gcd(a,b) = gcd(A,B).

effects and **post-condition:** $a = b$, $gcd(a,b) = gcd(A,B)$

An elaboration of :1.2: can be made if we observe that

$$a = gcd(A,B) \supset \textbf{true}$$

and

$$x = a \supset x = gcd(A,B)$$

But since :1.2: assumes $a = gcd(A,B)$ we can write

:1.2.1:

assumptions: write access to x is required, read access to a is required, neither multiplication nor division is to be used

:1.2.1: $x \leftarrow a$

effect and **post-condition:** $x = a$

As a result, the object/assumption table for :1.1, :1.2:, :1.1.1:, and :1.2.1: is

	0	1	2	3	4	5	6	7	8	9	10	11
:1.1:	1	1	1		1	1		♦			1	1
:1.2:	1			1		1	1		♦	♦		
:1.1.1:	1	1	1		1	1					1	1
:1.2.1:	1			1		1						

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) $a = \gcd(A,B)$
7) $(a = b) \wedge \gcd(A,B) = \gcd(a,b) \supset a = \gcd(A,B)$
8) $x = a \supset x = \gcd(A,B)$
9) $a = \gcd(A,B) \supset$ **true**
10) write access to b is required
11) read access to b is required

Adopting the convention that a' and b' equal the values of a and b just prior to the most recent execution of :1.1.1.2: we elaborate :1.1.1:

:1.1.1.1:

assumptions: $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $[(a < a' \vee b < b') \wedge \gcd(a,b) = \gcd(A,B) \wedge a \neq b) \supset (a \neq b \wedge \gcd(a,b) = \gcd(a',b')) \wedge \max(a',b') > \max(a,b)]$, $\gcd(a,b)=\gcd(A,B)$, neither multiplication nor division is to be used

:1.1.1.1: **while** $a \neq b$ **do**

:1.1.1.2:

assumptions: $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read and write access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $a \neq b$, $\gcd(a,b) = \gcd(A,B)$, neither multiplication nor division is to be used

:1.1.1.2: DECREASE EITHER a, b, OR BOTH a AND b SUCH THAT gcd(a,b) = gcd(A,B).

effect and
post-condition: $(a < a' \vee b < b')$, $\gcd(a,b) = \gcd(A,B)$

effects and
post-condition: $a = b$, $\gcd(a,b) = \gcd(A,B)$

The object/assumption table for :1.1.1.1: and :1.1.1.2: is

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
:1.1.1.1:	1	1	1			1						1	1	1	1	
:1.1.1.2:	1	1	1		1	1					1	1	1		1	1

1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
13) $(((a<a' \vee b<b') \wedge gcd(a,b)=gcd(A,B) \wedge a\neq b) \supset$
    $(a\neq b \wedge gcd(a,b) = gcd(A,B)) \wedge max(a',b') > max(a,b))$
14) $gcd(a,b) = gcd(A,B)$
15) $a \neq b$

If we add the following assumption to :1.1.1.2: we can "hide" A and B from :1.1.1.2.1:

$[(gcd(a,b)=gcd(A,B))\supset$
    $(gcd(a',b')=gcd(a,b)$, holds prior to executing :1.1.1.2.1:)] and

$[(gcd(a',b')=gcd(a,b)$, holds after executing :1.1.1.2.1:)$\supset$
    $(gcd(a,b)=gcd(A,B))]$

This is verifiable in the context of :1.1.1.2: and :1.1.1.1:.

:1.1.1.2.1:

assumptions: a > 0, b > 0, a and b are integer variables, A symbolizes the initial value of a and B symbolizes the initial value of b, read and write access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), neither multiplication nor division is to be used

:1.1.1.2.1: DECREASE EITHER a, b, OR BOTH a AND b SUCH THAT gcd(a,b) = gcd(a',b').

effects and **post-condition:** (a < a' or b < b'), gcd(a,b) = gcd(a',b')

The table for :1.1.1.2: and :1.1.1.2.1: is

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
:1.1.1.2:	1	1	1		1	1					1	1	1		1	1	+	
:1.1.1.2.1:	1	1			1	1					1	1	1			1		1

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
14) $gcd(a,b) = gcd(A,B)$
15) $a \neq b$
16) $[(gcd(a,b)=gcd(A,B)) \supset$
    $(gcd(a',b')=gcd(a,b)$, holds prior to executing :1.1.1.2.1:)] and
    [($gcd(a',b')=gcd(a,b)$, holds after executing :1.1.1.2.1:)$\supset$
    $(gcd(a,b)=gcd(A,B))]$
17) $gcd(a,b) = gcd(a',b')$

We can now elaborate :1.1.1.2.1: as

:1.1.1.2.1.1:

assumptions: $a > 0$, $b > 0$, a and b are integer variables, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $a \neq b$, $(a > b \wedge \gcd(a',b') = \gcd(a-b,b) \vee a \leq b \wedge \gcd(a',b') = \gcd(a,b-a)$, neither multiplication nor division is to be used

:1.1.1.2.1.1: if a > b **then**

:1.1.1.2.1.2:

assumptions: $\gcd(a',b') = \gcd(a-b,b)$, $a > 0$, $b > 0$, a and b are integer variables, write access to a is required, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, neither multiplication nor division is to be used

:1.1.1.2.1.2: DECREASE a BY b.

effect and
post-condition: a has been decreased by b.

else

:1.1.1.2.1.3:

assumptions: $\gcd(a',b') = \gcd(a,b-a)$, $a > 0$, $b > 0$, a and b are integer variables, read access to a is required, write access to b is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, neither multiplication nor division is to be used

:1.1.1.2.1.3: DECREASE b BY a.

effect and
post-condition: b has been decreased by a

post-condition: $\gcd(a,b) = \gcd(a',b')$, $(a < a'$ or $b < b')$

verification: given that $a \neq b$ and the other initial conditions the assumptions for :1.1.1.2.1.1: is a theorem from mathematics.

The object/assumption table for these parts is then:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
:1.1.1.2.1.1:	1	1				1						1	1			1			1		
:1.1.1.2.1.2:	1	1			1	1						1								1	
:1.1.1.2.1.3:	1	1				1					1	1									1

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:
15) $a \neq b$
18) $(a > b \wedge \gcd(a',b') = \gcd(a-b,b) \vee$
$a \leq b \wedge \gcd(a',b') = \gcd(a,b-a)$
19) $\gcd(a',b') = \gcd(a-b,b)$
20) $\gcd(a',b') = \gcd(a,b-a)$

Lastly, if we add

$$gcd(a',b') = gcd(a,b-a) \supset \textbf{true}$$

and

$$b \text{ decreased by } a \supset gcd(a,b)=gcd(a',b')$$

to the assumptions of :1.1.1.2.1.3: and

$$gcd(a',b')=gcd(a-b,b) \supset \textbf{true}$$

and

$$a \text{ decreased by } b \supset gcd(a,b)=gcd(a',b')$$

to the assumptions of :1.1.1.2.1.2:, we can elaborate these objects to

:1.1.1.2.1.2.1:

assumptions: write access to a is required, read access to a is required, read access to b is required, neither multiplication nor division is to be used

:1.1.1.2.1.2.1: a ← a - b.

effects and post-condition: a decreased by b

and

:1.1.1.2.1.3.1:

assumptions: read access to a is required, write access to b is required, read access to b is required, neither multiplication nor division is to be used

:1.1.1.2.1.3.1: b ← b - a.

effects and post-condition: b decreased by a

The relevant tables are:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
:1·1·1·2·1·2:	1	1			1	1						1	1							1		+	
:1·1·1·2·1·3:	1	1				1					1	1	1								1		+
:1·1·1·2·1·2·1:	1				1	1						1											
:1·1·1·2·1·3·1:	1					1					1	1											

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
19) gcd(a',b') = gcd(a-b,b)
20) gcd(a',b') = gcd(a,b-a)
21) gcd(a',b') = gcd(a,b-a) ⊃ **true**,
    b decreased by a ⊃ gcd(a',b') = gcd(a,b)
22) gcd(a',b') = gcd(a-b,b) ⊃ **true**,
    a decreased by b ⊃ gcd(a,b) = gcd(a',b')

Below is a display of the entire table. This table will be used in chapter IV to evaluate different decompositions of the program, and to observe where interactions were introduced as the program was developed.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
:1:	I	I	I	I	I	I					I	I											
:1·1:	I	I	I		I	I		+			I	I											
:1·2:	I			I		I	I		+	+													
:1·1·1:	I	I	I		I	I					I	I											
:1·2·1:	I			I		I																	
:1·1·1·1:	I	I	I			I						I	I	I	I								
:1·1·1·2:	I	I	I		I	I					I	I	I		I	I	+						
:1·1·1·2·1:	I	I			I	I					I	I	I			I		I					
:1·1·1·2·1·1:	I	I				I						I	I			I			I				
:1·1·1·2·1·2:	I	I			I	I						I	I							I		+	
:1·1·1·2·1·3:	I	I				I					I	I	I								I		+
:1·1·1·2·1·2·1:	I				I	I						I											
:1·1·1·2·1·3·1:	I					I					I	I											

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) a = gcd(A,B)
7) (a = b) ∧ gcd(A,B) = gcd(a,b) ⊃ a = gcd(A,B)
8) x = a ⊃ x = gcd(A,B)
9) a = gcd(A,B) ⊃ **true**
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
13) (((a<a' ∨ b<b') ∧ gcd(a,b)=gcd(A,B) ∧ a≠b) ⊃
    (a≠b ∧ gcd(a,b) = gcd(A,B)) ∧ max(a',b') > max(a,b))
14) gcd(a,b) = gcd(A,B)
15) a ≠ b
16) [(gcd(a,b)=gcd(A,B))⊃
    (gcd(a',b')=gcd(a,b), holds prior to executing :1.1.1.2.1:)] and
    [(gcd(a',b')=gcd(a,b), holds after executing :1.1.1.2.1:)⊃
    (gcd(a,b)=gcd(A,B))]
17) gcd(a,b) = gcd(a',b')
18) (a > b ∧ gcd(a',b') = gcd(a-b,b) ∨
    a ≤ b ∧ gcd(a',b') = gcd(a,b-a)
19) gcd(a',b') = gcd(a-b,b)
20) gcd(a',b') = gcd(a,b-a)
21) gcd(a',b') = gcd(a,b-a) ⊃ **true**,
    b decreased by a ⊃ gcd(a',b') = gcd(a,b)
22) gcd(a',b') = gcd(a-b,b) ⊃ **true**,
    a decreased by b ⊃ gcd(a,b) = gcd(a',b')

SUMMARY

The definition of program structure in Chapter I has been used to explicitly describe the kinds of assumptions which are associated with objects. These assumptions include the "pre-conditions", described by Dijkstra[DJ5], as well as assumptions about the capabilities an object must have in order to achieve its effects. Lastly, a program is developed in order to demonstrate the way assumptions are preserved and recorded. The object/assumption table for this program will be used in Chapter IV in order to examine various decompositions of the program in the light of the measure which is presented in Chapter III.

CHAPTER III

A MEASURE OF PROGRAM STRUCTURE

This chapter presents and justifies a calculation which is used to measure how much groups of objects in a program interact. Next, it is shown that the problem of finding an arrangement of objects which interacts least is tractable only for small programs. Instead, several theorems and heuristics are presented which provide useful guidelines for controlling interactions as a program is developed. Lastly, it is observed that certain assumptions should affect structure less than others. For example, assumptions which are not subject to change - theorems from mathematics - can safely be shared without influencing the difficulty of changing a program. Consequently, the definition of the measure is modified to incorporate this notion.

DEFINITION OF A MEASURE

Dictionary definitions of the word "measure" use phrases such as "reference standards to which something is valued", "a criterion", or "extent, degree or quantity". These phrases and the definition of program structure from chapter II suggest that a measure of program structure should provide a valuation of the interactions among identified groups of objects in a program. Van Emden[vE1,vE2] has described a calculation based on data contained in an object/assumption table which characterizes the degree to which collections of objects interact. The calculation is called **entropy loading**. Choosing this calculation as a measure of program structure is justified if we regard programs as complex systems which have identifiable objects which, in turn, possess properties - in this case, an object has the properties of making or not making certain assumptions. van Emden[vE2] has shown that entropy loading provides a measure of what Williams and Lambert[WL] call an "association". Informally, this means that the calculation indicates the degree to which designated collections of objects are similar to one another. Entropy loading measures the amount of information shared among collections of objects as opposed to the information used inside each collection. Alexander[AL] and Watanabe[WA] have also used similar techniques to analyze complex systems. In the present application, where object/assumption tables represent programs, designers can use the calculation as a guide for controlling interactions among collections of objects which are designated as subsystems.

En' [illegible] ɔading is defined for some subset of the rows, say S, of an object/assumption table containing n rows. Suppose that the objects in S are partitioned into two sets A and B such that $A \cup B = S$ and $A \cap B = \text{<empty>}$. The entropy loading of S for the partition into A and B is defined to be

$$C(S) = H(A) + H(B) - H(S)$$

where

$$H(X) = \sum_i (n_i/n) \log (n/n_i)$$

$$= \log n - (1/n) \sum_i n_i \log (n_i)$$

To determine the n_i's

(a) construct a table containing only those columns of S which contain at least one occurrence of a "1" in the subset of rows, X.

(b) regarding each row in this table as a one-zero pattern, the n_i's are the number of occurrences of each distinct pattern (Note that the sum of all n_i's equals n, the number of rows in S.)

For example, if a set of rows, S, is

$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 1 & 1 \\ 5 & 0 & 0 & 1 & 0 & 1 & 1 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Let A be the rows {1,2,3} and B the rows {4,5,6}, then $H(S) = H(A \cup B) = \log 6$ since at least one "1" occurs in every column, and each row is distinct. Similarly, H(A) can be computed by noting that at least one "1" occurs in A only in columns a, d, and f, and from

$$\begin{array}{|ccc} a & d & f \\ \hline 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{array}$$

1 0 0 occurs once; 0 0 1 occurs three times; 0 1 0 occurs once; and 0 0 0 occurs once, then

$$H(A) = \log 6 - (1/6)\ 3 \log 3 \qquad (*)$$

Similarly, since at least one "1" occurs in columns b, c, e, and f of B, 0 0 0 0 occurs twice; 0 0 0 1 occurs once; 1 0 1 1 occurs once; 0 1 1 1 occurs once; and 0 0 1 0 occurs once, ano

$$H(B) = \log 6 - (1/6) * 2 \log 2$$

Finally,

$$C(S) = C(A \cup B) = \log 6 - (1/6) * (2 \log 2 + 3 \log 3)$$

The calculation can be applied to A and B, and hence to any binary tree decomposition of S. That is, if $A = E \cup F$ and $E \cap F$ = <empty> then

$$C(A) = C(E \cup F) = H(E) + H(F) - H(A)$$

Note that H(A) is the same value as in (*), and that E and F define a set of columns from all the rows of the subset S. For example, if E is $\{1\}$ and F is $\{2,3\}$ then

$$H(E) = \log 6 - (1/6) (5 \log 5)$$

and

$$H(F) = \log 6 - (1/6) (2 \log 2 + 3 \log 3)$$

and

$$C(A) = C(E \cup F) = \log 6 - (1/6) (5 \log 5 + 2 \log 2)$$

Hence, an entropy loading value can be computed for each non-terminal node of a binary tree which represents some partition of S.

Entropy loading has several properties(van Emden[vE1]). The most important of these are

(1) C is always non-negative.

(2) If $S = A \cup B$, $A \cap B$ = <empty> and $S = E \cup F$, $E \cap F$ = <empty> and $C(A \cup B) < C(E \cup F)$ then A and B interact less than do E and F.

INTERPRETING THE MEASURE

Intuitively, entropy loading represents the extent to which information is shared between two groups of objects. Thus it characterizes a partition of the objects in a program, given the assumptions those objects make. As a measure of structure, this means that groups of objects having small entropy loadings possess better structure than other groups of those objects having larger entropy loadings - at least according to the definition of structure appearing in Chapter I.

A consequence of this property sometimes permits groups of objects that share the same number of assumptions to be distinguished. For example, except for the order of their rows, the two tables below are identical.

u	1010
v	0010
w	0110
x	0100
y	0001
z	1001

u	1010
v	0010
z	1001
w	0110
x	0100
y	0001

$$\begin{aligned} H(u,v,w,x,y,z) &= \log 6 \\ H(u,v,w) &= \log 6 \\ H(x,y,z) &= \log 6 - (1/3) \log 2 \\ C(\ (u,v,w),\ (x,y,z)\ ) &= \log 6 - (1/3) \log 2 \end{aligned}$$

$$\begin{aligned} H(u,v,z) &= \log 6 - (1/3) \log 2 \\ H(w,x,y) &= \log 6 - (2/3) \log 3 \\ C(\ (u,v,z),\ (w,x,y)\ ) &= \log 6 - \log 2 \end{aligned}$$

Objects (u,v,w) share two assumptions with objects (x,y,z) and (u,v,z) share two assumptions with (w,x,y). However, the entropy loading for ((u,v,z),(w,x,y)) is less than the entropy loading for ((u,v,w),(x,y,z)). This occurs because two assumptions are shared among u, v, z and one assumption is shared among w, x, y but only one assumption is shared among u, v, w and only one assumption is shared among x, y, z. Thus entropy loadings can often distinguish different decompositions even though the number of shared assumptions among the subsets of each decomposition is the same.

In one sense, the measure can also be used to compare different programs. Since a program is regarded as a partitioned collection of objects, where each object makes assumptions, any program represents a system whose structure can be measured. Hence, comparisons of entropy loadings for two different programs allow the structure implied by the partitions of the objects in each program to be compared. Using the measure to compare programs, however, does not seem to be useful unless the programs are related in some other way. For example, it is conceivable that a large program and a small program might be partitioned so that their entropy loadings are approximately the same. Such a comparison only indicates that the relative amounts of sharing in each program, for the selected partitions, are about the same. Analogously, a comparison of two programs in terms of the number of statements each contains depends on the way a statement is defined for each program. If in one program a statement is an assembly language instruction and in another a statement is a FORTRAN statement, the programs can be compared, but the comparison may not be very useful. Dijkstra[DJ1 in the section,"On Comparing Programs"] has cited similar difficulties for other kinds of comparisons except where a mapping can be found that associates the parts of the programs being compared. Such a mapping can be found if the programs that are compared represent successive stages in the step-wise construction of a program. One such mapping consists of associating an object with the objects into which it is refined. This mapping motivates the methods described in the next section.

These interpretations of the measure permit the definition of entropy loading to be extended in order to characterize a partition containing more than two sets of objects. If S is partitioned into n subsets, $s_1, s_2, \ldots, s_n$ then C can be defined as

$$C(s_1,s_2,\ldots,s_n) = (\textstyle\sum_i H(s_i)) - H(S).$$

Here, C is non-negative and indicates the amount of sharing inside the subsets relative to the amount of sharing among them. Note, however, that just a single figure characterizes this partition. Further, there seems to be no useful relationship between the values that result from applications of the original definition and this extended one.

APPLYING THE MEASURE TO OBJECT/ASSUMPTION TABLES FOR PROGRAMS

We propose to use the measure to suggest ways of controlling interactions as a program is developed. Elaborations which lead to high entropy loading values at late stages in the development of a program are to be avoided. This is demonstrated by the examples in Chapter IV.

We demonstrate several theorems which are relevant to fixed sets of rows from a table. These theorems justify several heuristics which are explained in this chapter and used in Chapter IV. The fixed sets of rows (the sets S used in the definition of the measure) will be **N-th stages** in the object/assumption table for a program.

> The **N-th stage** is defined to be all the objects whose <object name>'s consist of N integers separated by periods and those objects whose names consist of fewer than N integers but which will never be elaborated.

Informally, the N-th stages consist of the terminal nodes, at each elaboration, in the map for the development. For example, stage 1 of the development of the gcd computation consisted of object :1:; stage 2 consisted of objects :1.1: and :1.2:; stage 4 consisted of :1.1.1.1:, :1.1.1.2:, and :1.2.1:. After the theorems have been proved, several heuristics will be stated for controlling interactions by using a development at stage N-1 to suggest bounds for entropy loadings at stage N.

Theorem 1: Given:

(1) a collection of rows $S = A \cup B$ and $A \cap B = \text{<empty>}$ where A contains a rows and B contains b rows;

(2) S makes assumptions $P = C \cup D$ and $C \cap D = \text{<empty>}$;

(3) any row in A makes assumptions only in C and any row in B makes assumptions only in D;

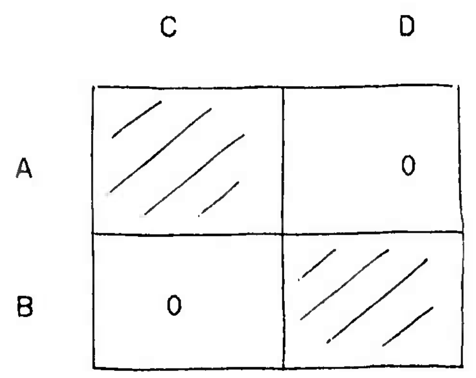
(4) $a \leq b$ and a is as small as possible for S, subject to (1), (2), and (3).

Conclude:

$C(A \cup B)$ achieves its minimum value for $S = A \cup B$ and

$$\log(a + b) - 1/(a + b) * (a \log a + b \log b)$$

Informally, the diagram shows the areas containing ones and possibly zeros as shaded, where A corresponds to some subset of rows in S which contains as few rows as possible.



Proof:

$$H(A) = \log(a + b) - (1 / (a + b)) (b \log b + \sum_i a_i \log a_i)$$

where $\sum_i a_i = a$ and the subscripted a's correspond to the partition of A imposed by C. Similarly

$$H(B) = \log(a + b) - (1 / (a + b)) (a \log a + \sum_i b_i \log b_i)$$

where $\sum_i b_i = b$ and the subscripted b's correspond to the partition of B imposed by D. Lastly,

$$H(A \cup B) = \log (a + b) - (1 / (a + b)) (\sum_i a_i \log a_i + \sum_i b_i \log b_i)$$

and

$$H(A) + H(B) - H(A \cup B) = \log (a + b) - (1 / (a + b) (a \log a + b \log b)$$

Now the restriction that a be as small as possible is necessary since

$\log(a+b)-(a \log a + b \log b)/(a+b) =$

$$(a/(a+b)) \log((a+b)/a) + (b/(a+b)) \log((a+b)/b)$$

and for a and b positive integers, the right side reaches its maximum when a = b and its minimum for the smallest a. Note that the roles of a and b can be reversed.//

Consequently, the best decompositions are known for tables that satisfy the conditions of the theorem. Although this theorem applies only to an entire table, analogous results can be obtained for decompositions of objects in A as well as B. Minimal entropy loadings occur for analogous partitions of A and B but their values cannot be derived directly from the above theorem. Another theorem states that objects which make identical assumptions should occur in the same subset of a partition.

Theorem 2: Given:

(1) a collection of s rows, $S = A \cup B$ and $A \cap B$ = <empty> where A contains a rows and B contains b rows and $a + b = s$;

(2) $a \geq b$;

(3) there is a collection of P rows which is a subset of S, containing p rows, $p > 1$ and $p < s$, which make identical assumptions;

(4) $P \cap A$ and $P \cap B$ are both non-empty;

Conclude:

There exist A' and B' such that $A' \cup B' = S$, $A' \cap B'$ = <empty> and $A' = A \cup P$ and $B' = B - P$ and

$$C(A \cup B) \geq C(A' \cup B')$$

Proof: Since $C(A \cup B)$ and $C(A' \cup B')$ contain a term which is identical, namely $H(A \cup B)$, if we show that $H(A') + H(B') \leq H(A) + H(B)$ then the theorem is proved.

Consider first the expansion of H(A), i.e.

$$H(A) = \log (a + b) - (1/(a + b)) (R + (p + t) \log (p + t))$$

where R is the sum of those terms describing subsets of a partition, but excluding the subset which contains the P identical rows. Here, $p + t$ equals the number of rows contained in the subset of the partition induced by A but containing P. Note that $t \geq 0$ and is just the difference between the number of rows in the subset of the partition induced by A, but containing P, and p.

Similarly

$$H(B) = \log (a + b) - (1/(a + b)) (T + (p + u) \log (p + u))$$

where T and u play analogous roles to R and t. Now, $H(A) = H(A')$ since the partition determined by A' is identical to the partition determined by A. But,

$$H(B') = \log (a + b) - (1/(a + b)) (T' + (p + u') \log (p + u')).$$

We must show that

$$(1/(a + b)) (T' + (p + u') \log (p + u')) \geq$$
$$(1/(a + b)) (T + (p + u) \log (p + u))$$

The two sides are equal if the assumptions made by B' are identical to those made by B. Further, if the assumptions made by B' are not the same as those made by B, then B' makes **fewer** assumptions than B. In this last case, there are a finite number of terms of T' which correspond to subsets of a partition with respect to B' which contain fewer items than the corresponding subsets for a partition with respect to B. This means that $u' \geq u$ and for some finite number of terms in T, there are identical terms in T' and there are terms in T' greater than the corresponding ones in T. We then must show that after excluding the identical terms in T and T' that the remaining terms in (*1): T' + (p + u') log (p + u') are greater than or equal to the remaining terms in (*2): T + (p + u) log (p + u). Denote by

$$\sum_j q_j \log q_j$$

the remaining terms in (*1). Denote by

$$\sum_i r_i \log r_i$$

the remaining terms in (*2), where

$$\sum_i r_i = \sum_j q_j.$$

Note that there are fewer q_j's than r_i's and that each q_j is the sum of several of the r_i's which correspond to the subset of rows which q_i designates. This means that if

$$\sum_k c_k \log c_k \leq (\sum_k c_k) \log (\sum_k c_k)$$

then

$$\sum_i r_i \log r_i \leq \sum_j q_j \log q_j$$

and the theorem is proved.

But, $\sum_k c_k \log c_k \leq (\sum_k c_k) \log (\sum_k c_k)$ since for $c_k \geq 1$ it is just the logarithms of both sides of the inequality

$$_k (c_k \uparrow c_k) \leq (\sum_k c_k) \uparrow (\sum_k c_k)$$

(where "↑" indicates exponentiation). //

Theorem 3: If a set of rows S satisfies the following conditions:

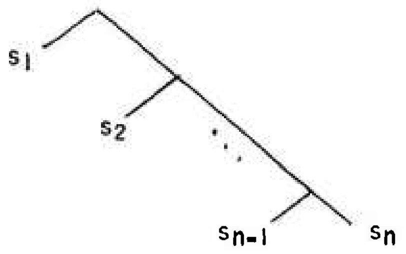
(a) S makes k assumptions ($p_1,p_2,p_3,\ldots,p_k$);

(b) S is partitioned into n sets such that there are q_1 elements in the first, q_2 in the second,..., q_n in the n-th set, where each set is denoted by s_i, $1 \leq i \leq n$;

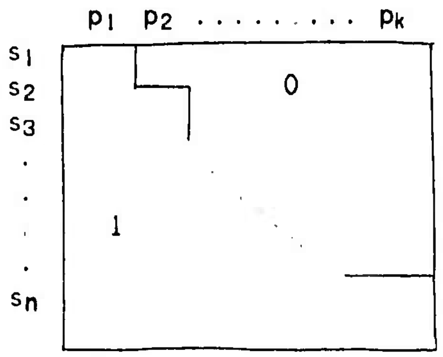
(c) the rows in a particular s_i are identical;

(d) p_j is made by s_i, $i \le j \le k$

then the minimal entropy loadings are achieved for a decomposition of the form



A table which illustrates the conditions of the theorem is



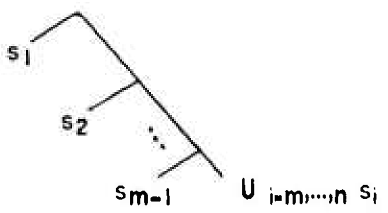
Proof: Assume that $\sum_{i=1,2,\dots,n} q_i = M$. The proof is by induction on m which indicates the subsets s_i of S. Observe that if m = 1

$$H(s_1) = \log M - (1/M)\,(M \log M) = 0$$

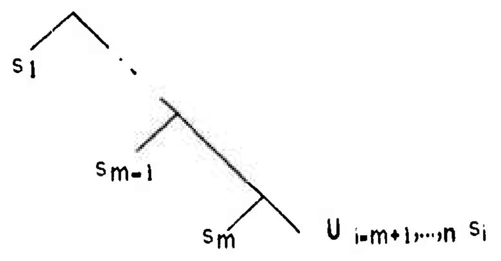
and that

$$H(\cup_{i=1,\dots,n} s_i) = H(S).$$

Hence $C(s_1 \cup (\cup_{i=2,\dots,n} s_i)) = 0$, which is the minimum value which any decomposition could have. (Here, U means "the union of".) Assume that for $j < m$, the entropy loadings for the partition of S indicated by



are as small as possible. We wish to show that the entropy loadings for



are as small as possible. Note that

$$H(s_{m+1}) = \log M - (1/M) \left(\sum_{i=1,\ldots,m} q_i \log q_i + \left(\sum_{i=m+1,\ldots,n} q_i \right) \log \left(\sum_{i=m+1,\ldots,n} q_i \right) \right)$$

and that

$$H(\cup_{i=m+2,\ldots,n} s_i) = H(S)$$

Suppose there is an A and B such that $A \cup B = \cup_{i=m,\ldots,n} q_i$ and $A \cap B =$ <empty> but that

$$(*): \quad C(A \cup B) < C(s_{m+1} \cup (\cup_{i=m+2,\ldots,n} s_i))$$

Clearly a row of s_n makes all the k assumptions and if at least one row of s_n is in A and at least one row of s_n is in B then

$$C(A \cup B) = \log M - (1/M) (\textstyle\sum_{i=1,\ldots,n} q_i \log q_i) > C(s_{m+1} \cup (\cup_{i=m+2,\ldots,n} s_i))$$

Hence, s_n must be in either B or A, say B, and $C(A \cup B) = H(A)$. But the smallest such value equals $H(s_{m+1})$, which contradicts the assumption (*). Therefore, by the principle of mathematical induction, the theorem is proved.//

Several observations regarding entropy loading can now be justified:

(1) Theorem 1 describes a situation for which the best entropy loading values are known. If programs are constructed with these properties, we know how they should be decomposed.

(2) Theorem 2 suggests that objects which make identical assumptions should occur in the same portions of a decomposition.

(3) Theorem 3 provides a best decomposition for a situation where different objects actually share several assumptions.

THE CLUSTERING PROBLEM

It now seems reasonable to ask whether there are ways of finding the best decomposition of a given N-th stage. If this were possible, we could observe whether best decompositions of early stages are really borne out by a development. If at any stage of finding a best decomposition, the set to be decomposed contains N items, then

$$\sum_{i=1,\dots,m}[N! / (i! * (N - i)!)] + X$$

(where m = N/2 and X is N! / ((N/2)! * (N/2)!) / 2 if N is even and N! /((N/2)! * (N - N/2)!) if N is odd)

partitions can be examined, and the best entropy loading value and its corresponding partition retained. Except for small N, this calculation is intractable. For this reason, several authors have attempted to find ways of determining best decompositions - "clusterings" - without having to examine all the partitions. vanEmden[vE1] has mentioned some of these methods, and has shown that the most popular of them are at best heuristics. Each fails for trivial object/assumption tables.

Is it reasonable to look for a tractable algorithm which can find a best decomposition? (The next paragraphs may be skipped at a first reading. The important result is simply that an affirmative answer to the question would be **very** surprising.)

* * *

As of this writing, the answer to the above question is not known. However, a particular recognition problem provides evidence to suspect that there is no such tractable algorithm. To describe this recognition problem, we present several definitions and theorems due to Cook[COO] and Karp[KA1].

Definition 1: P is the class of languages recognizable by one-tape Turing machines which operate in polynomial time.

Definition 2: π is the class of functions from Σ^* into Σ^* defined by one-tape Turing machines which operate in polynomial time.

Definition 3: Let L and M be languages. Then $L \propto M$ (L **is reducible to** M) if there is a function $f \in \pi$ such that $f(x) \in M$ if and only if $x \in L$.

Lemma 1 (Karp): If $L \propto M$ and $M \in P$ then $L \in P$.

Let P_2 denote the class of subsets of $\Sigma^* \times \Sigma^*$ which are recognizable in polynomial time. Given $L_2 \in P_2$ and a polynomial p, we define L as:

$$L = \{x \mid \text{there exists } y \text{ such that } \langle x,y\rangle \in L_2 \text{ and } \log(y) \leq p(\log(x))\}$$

L is said to be derived from L_2 by **polynomial-bounded existential quantification** and NP is the set of languages derived from P_2 by polynomial-bounded existential quantification, i.e. NP can be thought of as the set of languages which are recognizable, non-deterministically in polynomial time.

Now define the **satisfiability** problem

SATISFIABILITY
INPUT: c1 and c2 and ... cp (in conjunctive normal form)
PROPERTY: The conjunction of the given clauses is satisfiable; i.e. there is a set
$S \subset \{x_1,x_2,x_3,\ldots,x_n;x_1,x_2,x_3,\ldots,x_n\}$
such that

(a) S does not contain a complementary pair of literals
(b) $S \cap c_k \neq$ <empty>, $k=1,2,\ldots,p$.

Theorem(Cook): If $L \in NP$ then $L \propto$ SATISFIABILITY.

Corollary: P = NP if and only if SATISFIABILITY $\in$ P.

Karp has shown that a large number of problems can play the role of SAISFIABILITY in the above theorem. Such problems are called **complete**

Definition 4: The language L is (polynomial) **complete** if

a) $L \in NP$
b) SATISFIABILITY $\propto$ L

Theorem (Karp): Either all complete languages are in P, or none of them are. The former alternative holds if and only if P = NP.

Theorem (Karp): SATISFIABILITY $\propto$ PARTITION where PARTITION is defined as

INPUT: $(c_1,c_2,c_3,\ldots,c_m) \in Z^m$, positive integer m-tuples.

PROPERTY: There is a set $I \subset \{1,2,3,\ldots,m\}$ such that $\sum c_i = \sum c_j$ where each i is an element of I and each j is not an element of I.

It would be surprising, indeed, if all the complete problems were in P. We now show that a recognition problem related to the measure is at least complete.

Theorem: PARTITION $\propto$ LOG 2 CLUSTERING

where LOG 2 CLUSTERING is defined to be

INPUT: a (0-1) matrix, S, having N rows and M columns.

PROPERTY: there exists a clustering $A \cup B = S$, $A \cap B =$ <empty> such that $C(A \cup B) = \log 2$.

Proof: Let $N = \sum_i c_i$

Let $M = m$

$$S[I,J] = \begin{cases} 1, \sum_{k=1,\dots,J-1} c_k < I < \sum_{k=1,\dots,J+1} c_k, \\ \text{where an, upper bound for a summation of 0} \\ \text{equals 0 and an upper bound greater} \\ \text{than } N \text{ means } N+1. \\ \\ 0, \text{ otherwise.} \end{cases}$$

Now, if $(c_1, c_2, \dots, c_m)$ has a partition with the desired properties then we choose as a set A, all those rows corresponding to the c's in a single set of the partiton; the rest of the rows constituting B. Then

$H(A) = \log N - (1/N) [\sum_k c_k \log c_k] - (1/2) \log (N/2)$

where k is an element of I.

$H(B) = \log N - (1/N) [\sum_k c_k \log c_k] - (1/2) \log (N/2)$

where k is not an element of I.

$H(A \cup B) = \log N - (1/N) [\sum_k c_k \log c_k + \sum_j c_j \log c_j]$

where k is an element of I but where j is not an element of I.

Hence, $C(A \cup B) = \log N - \log N/2 = \log 2$. For all other matrices of the form described at the beginning of the proof, there is no partition which leads to a C value of log 2.//

Intuitively, this problem is not as difficult as the general clustering problem, yet a solution to the clustering problem, does not solve the partition problem, for the encoding above.

* * *

In short, a tractable solution to the clustering problem would be very surprising.

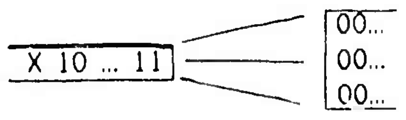
HEURISTICS FOR USING THE MEASURE

But just a solution to the clustering problem for an object/assumption table whose objects have many interactions really is not of much interest. Making the best of a bad design job is not of much interest. Programs should instead be composed of parts which interact very little. As programmers, we can modify the tables which represent our programs in order to improve their structure. This is not the case for data in an ecological study, for example, where an object/assumption table is fixed and a best decomposition might be of interest.

In chapter IV, the example developed in chapter II will be examined using of the measure. The measure is used in two ways. First, at early stages in a development the best decompositions are found. In most cases this is tractable because the number of objects is small. Second, if entropy loading figures for an elaboration of the objects in a best decomposition seem too large, attempts are made to modify the object/assumption table and the program so that the good properties of the earlier decomposition are preserved. The heuristics used are summarized below:

Assume we have a decomposition at the N-th stage in which the parts of the decomposition interact little, and that an elaboration has been made to the (N+1)-st stage.

> (1) Compute as a **rough lower bound**, RLB, for the entropy loading at the (N+1)-st stage, the entropy loading with respect to the N-th stage decomposition, where each object which was elaborated is replaced by as many new objects as appear in the (N+1)-st stage such that each object makes NO assumptions.
>
> For example, if the table below represents part of a table to the N-th stage and at the (N+1)-st stage the object X is elaborated to three objects, we compute the entropy loadings with respect to the decomposition at the N-th stage, but replacing X by three rows of zeros.



stage N stage (N+1)

> This computation takes into account the increased size of the table, but introduces new objects which do not interact at all.

(2) As a **rough upper bound**, RUB, for the entropy loadings at the (N+1)-st stage, compute the entropy loadings with respect to the N-th stage decompostion where each object which was elaborated is replaced by as many new objects as appear in the (N+1)-st stage, such that each new object is identical with its parent.

These guides are only rough indicators. The reader can generate simple situations where expansions lead to smaller or larger entropy loading values than those indicated. For many cases, however, they are quite useful.

(3) Now compute the entropy loadings of the (N+1)-st stage and compare its value with RLB and RUB. We clearly wish to make each elaboration so that the entropy loadings are as close to RLB as possible.

(4) If entropy loadings are greater than RUB, then several cases arise which involve interactions resulting from one or several of the following

(a) new assumptions not appearing at the N-th stage

(b) assumptions which appear at the N-th stage and are shared by the parents of subsets for which RUB is exceeded

There are several possible actions

(a) Accept the interactions that are present and proceed with the development.

(2) Attempt to localize the assumptions not appearing at the N-th stage to a single subset of the decomposition at the (N+1)-st stage. This implies that a new structure, for which the new objects do not have a parent, has been imposed on the (N+1)-st stage. An attempt should be made to place these objects in some subset where, except for the new assumptions, as much information as possible is shared. (This technique is consistent with Theorem 2.)

Chapter IV will display several examples where this technique can and cannot be used.

(3) Attempt to find some other decomposition of the objects. Hopefully, such a choice will result in few changes in the original decomposition.

SATURATION IN OBJECT/ASSUMPTION TABLES

It should be noted that the suggestions in the last section are only heuristics which can aid in evaluating the goodness of a decomposition. One situation which occurred when several examples were analyzed will be called **saturation** in an object/assumption table for a decomposition. Stated simply, this situation occurs whenever a table is decomposed to a depth where further attempts at decomposing subsets of certain objects results in identical entropy loading values for all possible decompositions. In a saturated table, the measure provides no help in distinguishing decompositions. Saturation occurs most frequently whenever a small number of objects together make many assumptions. Frequently, further refinements of objects lead to tables which are not saturated for a decomposition. This seems to occur whenever existing assumptions become localized to a small number of objects relative to the total number at a given stage. Examples of this situation are cited in the next chapter. Saturation indicates bad structure in the sense that all the objects at a particular stage share much information.

ON THE PROBABILITY OF CHANGE OF ASSUMPTIONS

The examples in Chapter II indicate that objects make many assumptions even in small programs. Yet, some of the assumptions are never likely to change even after major modifications of the program. Intuitively, these assumptions seem to have less influence on the structure of a program from one modification to the next than do assumptions which are very likely to change. Theorems from mathematics have already been cited as assumptions which can never be wrong and hence have zero probability of change. However, assumptions based on say "the position of information in a control block", which have a very high probability of change, have a great influence on structure from modification to modification. Theoretically, every assumption could be rated with a probability of change, even if that rating is simply a relative one, e.g. that one set of assumptions has a greater probability of change than another.

In this section, we extend the measure to take into account a probability of change for each assumption. This extension is shown to be consistent with the definition of entropy loading.

First, note the property:

> If an assumption, say X, is shared by every (no) row in some set of rows then for any decomposition of that set, the entropy loadings computed when considering X are identical with the entropy loadings computed when not considering X.

Proof: Just observe that any partition determined by a set of assumptions other than X is not changed if X is added to the set of assumptions.

Next, observe that no matter how many objects make assumptions which can never change or regardless of where those assumptions are made, there will be no ill effects because of those assumptions, should the program ever be changed. For this reason, unchanging information can be distributed without affecting the ease of either maintainence or change. However, these assumptions are relevant to understanding the program and should be listed in a specification or at least in the object/assumption table. Consequently, we modify the entropy loading calculation so that it remains consistent with the properties stated at the beginning of the chapter, but allows assumptions which are certain to change (probability of change = 1) to have the same effects as before and permits assumptions which will never change to exert no effects. The modification also has the property that it can be computed for any set of assumptions having any probability of change values associated with each assumption.

The only change involves the way H(X) is computed. As before,

$$H(X) = \log N - (1/N) \sum_i n_i \log n_i$$

$$= \sum_i (n_i / N) \log (N/n_i) \qquad (*)$$

where the n_i's correspond to the number of rows in each subset of a partition imposed by the assumptions of X on the N rows. (*) above is the standard definition of entropy from information theory - and has all its properties - if we interpret n_i/N as a probability so long as

$$\sum_i (n_i / N) = 1.$$

From this point of view, n_i need not be an integer. Instead, we compute the n_i's as follows.

(1) Choose a function of one variable, f, defined on the closed interval [0,1] such that $f(x) \geq x$ and f is strictly monotonic (such a function is just the probability of change itself).

(2) Construct a new table as follows:

(a) fill the table with those columns from the original table whose probability of change is 1 and associate with each of the N rows a value, $w_i = 1$ for the i-th row. (if the table has no such columns, create a new table with N rows, $w_i = 1$, $k_i = i$ and containing a single column each entry of which equals 1)

So long as there are assumptions which allow (b) to be executed, repeat (b), (c), and (d).

(b) Select a single assumption, say P with probability of change p, from those that remain (ignoring all assumptions whose probability of change equals zero), attach this column to the new table, and select the value of the parameter, π, as follows:

if the number of occurrences of "1" is less than N/2 in the column for P, then let π be 1; otherwise 0.

(c) for each row of the table which does not contain an entry for assumption P, say j, choose its entry to be identical with the entry for P in row i, $1 \leq i \leq N$, such that $k_i = k_j$. (this step will not need to be performed the first time (b) is executed)

(d) For each row in the new table, say i, which has an entry for P which equals π, add a new row, j, to the table which is identical to row i except that the entry for P is 1 - π and set w_j to w_i - f(p) * w_i and w_i equal to f(p) * w_i.

(3) Now for each pattern of 1's and 0's, sum the respective w's to get a set of s_j's and compute H(X) as

$$H(X) = \sum_j (s_j/N) \log (N/s_j) \text{ as } H(X)$$

The result of this computation is independent of the order in which the assumptions are used in the algorithm. The value is consistent with the observations stated earlier. The improvements of Chapter IV will use these ideas as a primary justification for ignoring certain assumptions.

An example of this computation is indicated by the table below:

1	.5	.25
0	1	1
1	0	1
1	1	0
1	0	1
0	0	1

where the value above each column represents the probability of change of the assumption associated with that column (this value will be used as f).

After step (a), the new table is

k	w	1
1	1	0
2	1	1
3	1	1
4	1	1
5	1	0

After selecting a remaining assumption, and executing b

$\pi = 1$ and

k	w	1	.5
1	1	0	1
2	1	1	0
3	1	1	1
4	1	1	0
5	1	0	0

(c) need not be executed, but after (d), the table is

k	w	1	.5
1	.5	0	1
2	1	1	0
3	.5	1	1
4	1	1	0
5	i	0	0
1	.5	0	0
3	.5	1	0

Next, (b) is executed leaving $\pi = 0$ and the table

k	w	1	.5	.25
1	.5	0	1	1
2	1	1	0	1
3	.5	1	1	0
4	1	1	0	ı
5	1	0	0	1
1	.5	0	0	
3	.5	1	0	

After executing (c), the table is

k	w	1	.5	.25
1	.5	0	1	1
2	1	1	0	1
3	.5	1	1	0
4	1	1	0	1
5	1	0	0	1
1	.5	0	0	1
3	.5	1	0	0

Lastly, after executing (d) the table is

k	w	1	.5	.25
1	.5	0	1	1
2	1	1	0	1
3	.125	1	1	0
4	1	1	0	1
5	1	0	0	1
1	.5	0	0	1
3	.125	1	0	0
3	.375	1	1	1
3	.375	1	0	1

Hence, H for this table can be computed from

011	.5
101	2.375
110	.125
001	1.5
100	.125
111	.375

The effect of this calculation has been to modify the influences of assumptions whenever entropy loading calculations are computed. Consequently, this same calculation can be used to modify the influences of assumptions for other properties besides "probability of change". For example, if the relation "importance of assumptions" can be established for a program, this modified calculation can be used.

CHAPTER IV

USING THE MEASURE

The heuristics described in Chapter III which provide suggested bounds for entropy loading values are used as guides to control interactions between the parts of a decomposition. Entropy loading values for various decompostions of the example from Chapter II are displayed. In addition, several other examples are developed using the measure as a guide.

INTRODUCTION

Chapters I and II provided a definition of program structure along with techniques for preserving assumptions which help to determine structure. Chapter III described a numerical calculation - entropy loading - which can use these assumptions to compare the "goodness" of different guesses at what the components of a decomposition of a program are. The theorems in Chapter III form the basis for methods of constructing elaborations for components in an initial decomposition. These methods are used to insure that components interact little in the final program, either by verifying the presence of reasonable interactions or by indicating that elaborations which interact less should be sought. Such programs have a good chance of satisfying the properties stated in the Introduction to this thesis.

These methods are first demonstrated with respect to the GCD example from Chapter II.

A GCD COMPUTATION

Despite the small size of the GCD program, entropy loading calculations can be used to show that certain decompositions of the objects have greater interactions between their parts than others. (For the following discussion, it is assumed that the probability of change of all assumptions is one, i.e. that all assumptions are likely to change and have potentially the same influence on structure.) Below is the object/assumption table for the GCD computation from Chapter II.

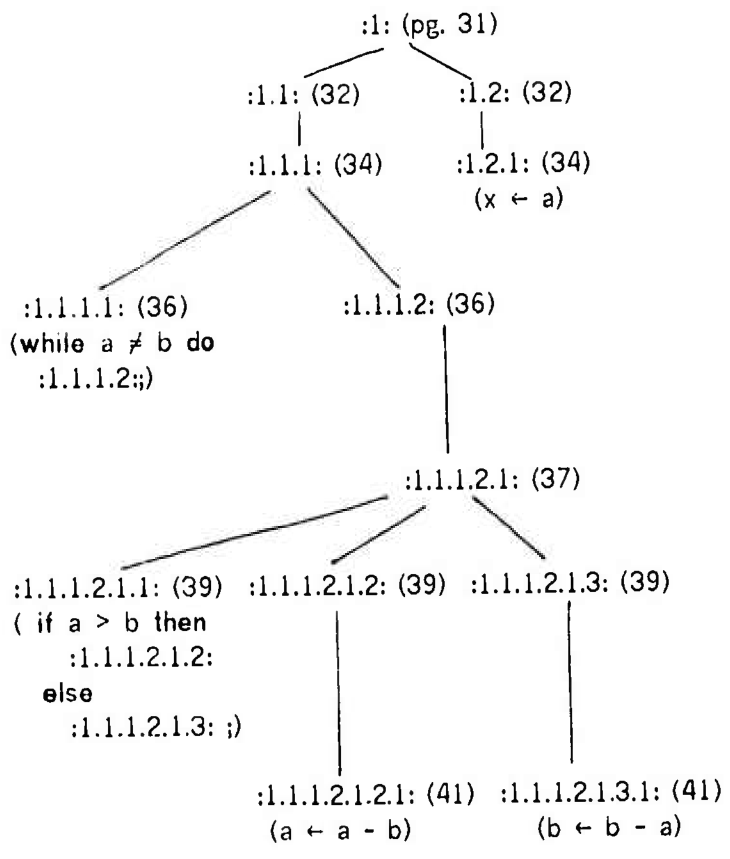
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
:1:	1	1	1	1	1	1					1	1											
:1·1:	1	1	1		1	1		◆			1	1											
:1·2:	1			1		1	1		◆	◆													
:1·1·1:	1	1	1		1	1					1	1											
:1·2·1:	1			1		1																	
:1·1·1·1:	1	1	1			1						1	1	1	1								
:1·1·1·2:	1	1	1		1	1					1	1	1		1	1	◆						
:1·1·1·2·1:	1	1			1	1					1	1	1			1		1					
:1·1·1·2·1·1:	1	1				1						1	1			1			1				
:1·1·1·2·1·2:	1	1			1	1						1	1							1		◆	
:1·1·1·2·1·3:	1	1				1					1	1	1								1		◆
:1·1·1·2·1·2·1:	1				1	1						1											
:1·1·1·2·1·3·1:	1					1					1	1											

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) $a = \gcd(A,B)$
7) $(a = b) \wedge \gcd(A,B) = \gcd(a,b) \supset a = \gcd(A,B)$
8) $x = a \supset x = \gcd(A,B)$
9) $a = \gcd(A,B) \supset$ **true**

10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:
13) $(((a<a' \vee b<b') \wedge gcd(a,b)=gcd(A,B) \wedge a \neq b) \supset (a \neq b \wedge gcd(a,b) = gcd(A,B)) \wedge max(a',b') > max(a,b))$
14) $gcd(a,b) = gcd(A,B)$
15) $a \neq b$
16) [$(gcd(a,b)=gcd(A,B)) \supset$ ($gcd(a',b')=gcd(a,b)$, holds prior to executing :1.1.1.2.1:)] and
[($gcd(a',b')=gcd(a,b)$, holds after executing :1.1.1.2.1:)$\supset$
($gcd(a,b)=gcd(A,B)$)]
17) $gcd(a,b) = gcd(a',b')$
18) $(a > b \wedge gcd(a',b') = gcd(a-b,b) \vee$
$a \leq b \wedge gcd(a',b') = gcd(a,b-a)$
19) $gcd(a',b') = gcd(a-b,b)$
20) $gcd(a',b') = gcd(a,b-a)$
21) $gcd(a',b') = gcd(a,b-a) \supset$ **true**,
b decreased by a $\supset gcd(a',b') = gcd(a,b)$
22) $gcd(a',b') = gcd(a-b,b) \supset$ **true**,
a decreased by b $\supset gcd(a,b) = gcd(a',b')$

The map for the development is

A GCD COMPUTATION



Object :1: assumes that positive integer values are contained in variables a and b and has the effect of assigning to x the value of the greatest common divisor of the initial contents of a and b (symbolized by A and B, respectively).

Object :1.1: leaves the value of gcd(a,b) in a and object :1.2: assigns the value of a to x.

The theorem that $(\text{gcd}(a,b) = \text{gcd}(A,B) \wedge a = b) \supset a = \text{gcd}(A,B)$ is hidden from :1.1.1: and $a = \text{gcd}(A,B)$ is hidden from :1.2.1:.

Object :1.1.1.1: controls the execution of :1.1.1.2: which maintains the invariant $(a < a' \vee b < b') \wedge \text{gcd}(a,b) = \text{gcd}(A,B)$, where a' and b' are the values of a and b prior to each iteration.

Object :1.1.1.2.1: hides the invariant $\text{gcd}(a,b) = \text{gcd}(A,B)$.

Finally, objects :1.1.1.2.1.1:, :1.1.1.2.1.2:, and :1.1.1.2.1.3: implement a conditional statement which results in $(a < a' \vee b < b') \wedge \text{gcd}(a,b) = \text{gcd}(a',b')$.

Not surprisingly, the best[1] decomposition for the development

represented by the terminal nodes of the tree

$$\begin{tikzcd}[arrows=-]
 & & & :1: \arrow[dll] \arrow[dr] & \\
 & :1.1: \arrow[d] & & & :1.2: \arrow[d] \\
 & :1.1.1: \arrow[dl] \arrow[dr] & & & :1.2.1: \\
:1.1.1.1: & & :1.1.1.2: & &
\end{tikzcd}$$

is a decomposition into two parts: (:1.1.1.1:, :1.1.1.2:) (**while** $a \neq b$

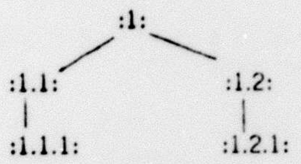
do ... , decrease a, b or both so that $\text{gcd}(a,b) = \text{gcd}(A,B)$) and

(:1.2.1:) ($x \leftarrow a$)

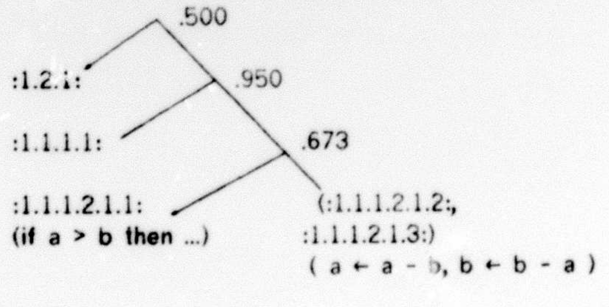
- - - - - - - - - - - - -

[1] Of two decompositions for the same objects, one decomposition is said to be "better" than another if, starting at the root, a difference in entropy loading values is found at some node and that decomposition has the smaller entropy loading value at that node. Clearly, this comparison is only meaningful for decompositions which are based on trees which have the same shape to the stage where they are compared.

with entropy loading value .637[2] which is identical with the values of the rough lower bound (RLB) and rough upper bound (RUB) for the expansion from



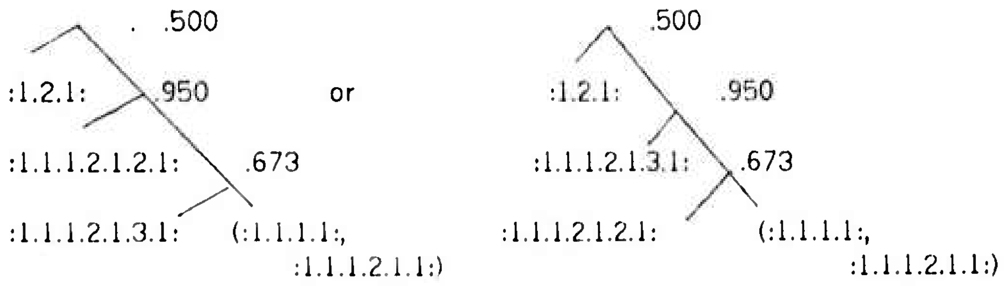
This indicates that the assignment of a to x interacts least. Similarly, after refining :1.1.1.2: to :1.1.1.2.1: the entropy loading of (:1.1.1.1:,:1.1.1.2.1:) and (:1.2.1:) is again equal to .637. Refinements of :1.1.1.2.1:, however, show that RLB and RUB are indeed "rough" bounds since for (:1.1.1.1:, :1.1.1.2.1:) and (:1.2.1:), RLB is .951 and RUB is 1.33, but the actual entropy loading for (*): (:1.1.1.1:, :1.1.1.2.1.1:, :1.1.1.2.1.2:, :1.1.1.2.1.3:) and (:1.2.1:) is .500, indicating that the distribution of the increased number of assumptions in the new table causes less interaction between the two subsets of the partition. The best decomposition of (*) appears below, with the entropy loading values at each non-terminal node.



[2] The numerical values representing entropy loadings are all computed with respect to natural logarithms and not logarithms to the base 2.

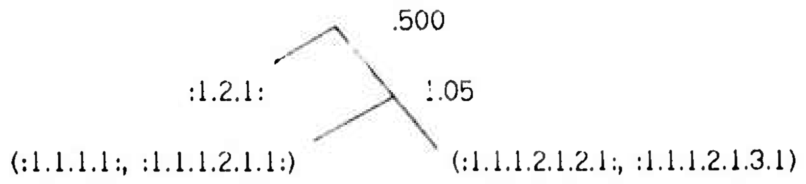
However, the refinements of :1.1.1.2.1.2: and :1.1.1.2.1.3: lead to a program whose best decomposition is

(A)



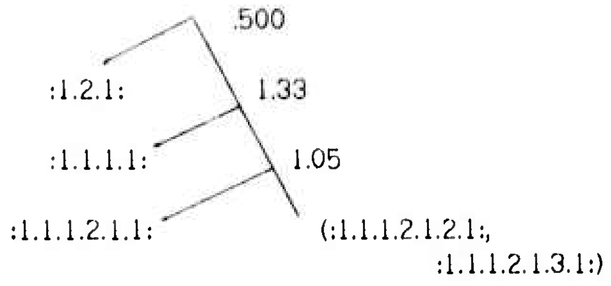
Further, the decomposition

(B)



is better than

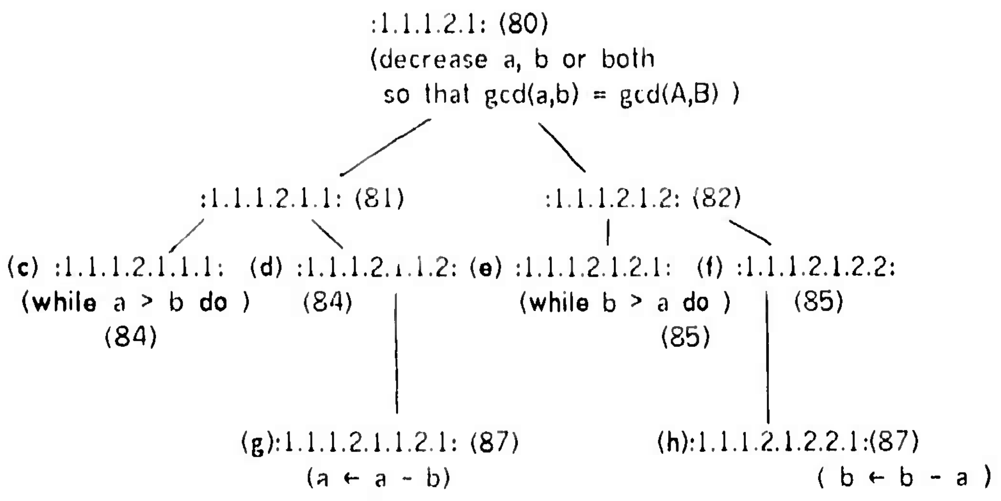
(C)



In particular, the best decomposition before the last refinement is not the best decomposition after the refinement. The reasons for this result are obvious from an examination of the object/assumption table (in the light of the above calculations, but are, perhaps, not so obvious without them). The assumptions made by :1.1.1.2.1.2: (decrease a by b) and :1.1.1.2.1.3: (decrease b by a) are more numerous than for their refinements. These additional assumptions increase the number of subsets in the partitions which determine the H values, but do so in a way which indicates little relative interaction among the subsets of the decomposition. However, since the final refinement involves few assumptions, there is greater relative interaction for the same decomposition. From the standpoint of interactions alone, the best decompositions are indicated in (A), and in a larger, but similar, example work assignments might be made based upon these decompositions. This decomposition separates the actions of assigning values to variables from the mechanisms which control these operations. Hence, the measure indicates that the control mechanisms interact most since they require more information about the program. However, should a designer wish to distribute more information, in the form of assumptions at earlier stages, the decompositions in (B) or (C) might be more appropriate.

A different elaboration of this version might arise by observing that for each execution of the body of the loop, two tests are made for each modification of either a or b.

GCD Computation (Version II)


:1.1.1.2.1: (80)
(decrease a, b or both so that gcd(a,b) = gcd(A,B))
:1.1.1.2.1.1: (81)
:1.1.1.2.1.2: (82)
(c) :1.1.1.2.1.1.1: (while a > b do) (84)
(d) :1.1.1.2.1.1.2: (84)
(e) :1.1.1.2.1.2.1: (while b > a do) (85)
(f) :1.1.1.2.1.2.2: (85)
(g):1.1.1.2.1.1.2.1: (87) (a ← a - b)
(h):1.1.1.2.1.2.2.1:(87) (b ← b - a)

The development of this version is identical with the development in Chapter II up to :1.1.1.2.1:. Version I maintains the invariant gcd(a,b) = gcd(a',b') but version II requires that a or b or both have been modified more than once, if possible.

Object :1.1.1.2.1.1: decreases a until it becomes smaller than b, but maintains the invariant gcd(a,b) = gcd(a',b').

Object :1.1.1.2.1.2: makes b smaller than a, but maintains the invariant gcd(a,b) = gcd(a',b').

Objects :1.1.1.2.1.1: and :1.1.1.2.1.2.1: are **while** constructions that control :1.1.1.2.1.1.2: and :1.1.1.2.1.2.2: respectively.

Lastly, :1.1.1.2.1.1.2.1: and :1.1.1.2.1.2.2.1: are elaborations from which are hidden information about the invariants that are being maintained.

A new elaboration of :1.1.1.2: might be

:1.1.1.2.1:

assumptions: neither multiplication nor division is to be used and a > 0, b > 0, a and b are integer variables, A symbolizes the initial value of a and B symbolizes the initial value of b, a is an integer variable, read and write access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(A,B) ≡ gcd(a',b') = gcd(a,b), gcd(a,b) = gcd(a',b'), :1.1.1.2.1: is the body of a loop which makes the test "a ≠ b"

:1.1.1.2.1: DECREASE EITHER a, b, OR BOTH a AND b SUCH THAT gcd(a,b) = gcd(a',b') WHERE a OR b HAVE BEEN MODIFIED MORE THAN ONCE, IF POSSIBLE.

effects and **post-condition:** (a < a' or b < b'), gcd(a,b) = gcd(a',b'), if possible, more than one modification of a or b should occur for each test of the outer loop.

The best decomposition for the three objects is

((:1.2.1:) (:1.1.1.1:, :1.1.1.2.1:)) .637 [1]

:1.1.1.2.1: is elaborated next

:1.1.1.2.1.1:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, write access to a is required, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), :1.1.1.2.1: is the body of a loop which makes the test "a $\neq$ b", neither multiplication nor division is to be used

:1.1.1.2.1.1: IF POSSIBLE, MAKE a SMALLER THAN b SUCH THAT gcd(a,b) = gcd(a',b')

effects and
post-condition: $a \leq a' \wedge gcd(a,b) = gcd(a',b') \wedge a'>b' \supset b' \geq a$

- - - - - - - - - - -

[1] This notation will be used instead of tree diagrams to indicate a decomposition and the entropy loading values for non-terminal nodes. (The value of an instance of <number> represents the entropy loading value for the parenthesized pair to its left.)

```
<decomposition> ::= ( <part list> <part list> ) <number> |
                    ( <part list> <part list> )

<part list>     ::= <decomposition> | <simple part list>

<simple part list> ::= <object name> |
                       <simple part list> , <object name>
```

:1.1.1.2.1.2:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, read access to a is required, write access to b is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $gcd(a,b) = gcd(a',b')$, neither multiplication nor division is to be used :1.1.1.2.1: is the body of a loop which makes the test "$a \neq b$"

:1.1.1.2.1.2: IF POSSIBLE, MAKE b SMALLER THAN a SUCH THAT gcd(a,b)=gcd(a',b')

effects and **post-condition:** $gcd(a,b) = gcd(a',b') \wedge b \leq b' \wedge (b' > a \supset a \geq b)$

verification: To show that the post-condition for :1.1.1.2.1: holds, i.e. $(a < a' \vee b < b') \wedge gcd(a,b) = gcd(a',b')$, if possible, more than one modification of a or b should occur for each test of the outer loop, holds.

Case 1: if $a' > b'$ then $b \geq a$ and $a < a'$, since after the execution of :1.1.1.2.1.2:, $gcd(a,b) = gcd(a',b')$.

Case 2: If $b' > a$ then $a \geq b$ which means that $b' > b$. But since $gcd(a,b) = gcd(a',b')$ the post-condition holds.

If more than one modification of either a or b can be made, this elaboration will make more than one such modification.

The relevant table is then

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
:1·1·1·2·1:	1	1			1	1					1	1	1			1		1	1
:1·1·1·2·1·1:	1	1			1	1						1	1					1	1
:1·1·1·2·1·2:	1	1				1					1	1	1					1	1

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
17) gcd(a,b) = gcd(a',b')
18) :1.1.1.2.1: is the body of a
    loop which makes the test "$a \neq b$"

RLB and RUB for this expansion both equal 1.04, but the best decomposition has better entropy loading values than RLB and RUB would suggest, and is derived from the last stage, i.e.

((:1.2.1:) ((:1.1.1.1:) (:1.1.1.2.1.1:, :1.1.1.2.1.2:)) 1.38) .562

Expanding :1.1.1.2.1.1: and :1.1.1.2.1.2: gives

(c) :1.1.1.2.1.1.1:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') $\wedge$ $a > b \supset$ gcd(a',b') = gcd(a-b,b), neither multiplication nor division is to be used

:1.1.1.2.1.1.1: **while** a > b **do**

(d) :1.1.1.2.1.1.2:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, write access to a is required, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') = gcd(a-b,b), neither multiplication nor division is to be used

:1.1.1.2.1.1.2: DECREASE a BY b

effect and
post-condition: a has been decreased by b.

effect and
post-condition: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, gcd(a',b') = gcd(a,b)

Similarly, we elaborate :1.1.1.2.1.2:

(e) :1.1.1.2.1.2.1:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), (gcd(a,b') $\wedge$ $b > a$) $\supset$ gcd(a',b') =gcd(a,b-a), neither multiplication nor division is to be used.

:1.1.1.2.1.2.1: **while** b > a **do**

(f) :1.1.1.2.1.2.2:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, read access to a is required, write access to b is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') = gcd(a,b-a), neither multiplication nor division is to be used

:1.1.1.2.1.2.2: DECREASE b BY a

effect and
post-condition: b has been decreased by a

effects and
post-condtion: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, gcd(a,b) = gcd(a',b')

The table for these two parts is then

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(c)	:1·1·1·2·1·1·1:	1	1				1						1	1					1						1	
(d)	:1·1·1·2·1·1·2:	1	1			1	1						1	1					1		1					
(e)	:1·1·1·2·1·2·1:	1	1				1						1	1					1							1
(f)	:1·1·1·2·1·2·2:	1	1				1					1	1	1					1			1				

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
prior to the last execution of :1.1.1.2:
17) $\gcd(a,b) = \gcd(a',b')$
19) $\gcd(a',b') = \gcd(a-b,b)$
20) $\gcd(a',b') = \gcd(a,b-a)$
23) $(\gcd(a',b') \wedge a > b) \supset \gcd(a',b') = \gcd(a-b,b)$
24) $(\gcd(a',b') \wedge b > a) \supset \gcd(a',b') = \gcd(a, b-a)$

For

((:1.2.1:) ((:1.1.1.1:)(:1.1.1.2.1.1:,:1.1.1.2.1.2:)) *1) *2

RLB is *1 = .868

RUB is *1 = .451 and *2 = .868

but the actual loadings for the above elaboration are

((:1.2.1) ((:1.1.1.1:) ((**c** , **d**) (**e** , **f**)) 1.33) .868) .451

Here the inner loops appear in two distinct subsets. It should be noted that if the **while** constructions appear together the entropy loading figures do not change, i.e. ((**c** , **e**) (**d** , **f**)) 1.33. Further,

((**c**) (**d** , **e** , **f**)) 1.24

This is the same entropy loading value for all other decompostitions of **c**, **d**, **e**, and **f** where one subset contains just a single object.

Lastly, the assignments can be elaborated if we add assumptions to the table as follows

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
(d)	:1.1.1.2.1.1.2:	I				I	I						!	I					I		I		♦	
(f)	:1.1.1.2.1.2.2:	I					I					I	I	I					I			I		♦

0) neither multiplication nor division is to be used
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
17) gcd(a,b) = gcd(a',b')
19) gcd(a',b') = gcd(a-b,b)
20) gcd(a',b') = gcd(a,b-a)
21) gcd(a',b') = gcd(a,b-a) ⊃ **true**,
    b decreased by a ⊃ gcd(a',b') = gcd(a,b)
22) gcd(a',b') = gcd(a-b,b) ⊃ **true**,
    a decreased by b ⊃ gcd(a,b) = gcd(a',b')

(g) :1.1.1.2.1.1.2.1:

assumptions: write access to a is required, read access to a is required, read access to b is required

:1.1.1.2.1.1.2.1: a ← a - b.

effects and
post-condition: a decreased by b

and

(h) :1.1.1.2.1.2.2.1:

assumptions: read access to a is required, write access to b is required, read access to b is required

:1.1.1.2.1.2.2.1: b ← b - a.

effects and
post-condition: b decreased by a

Now, note the following entropy loading calculations, where **a** names :1.2.1: and **b** names :1.1.1.1:,

(A): ((a) ((b) ((**c** , g) (**e** , **h**)) 1.79) 1.33) .451

but that

(B): ((a) ((b) ((**c** , **e**) (g , **h**)) 1.01) 1.33) .451

and

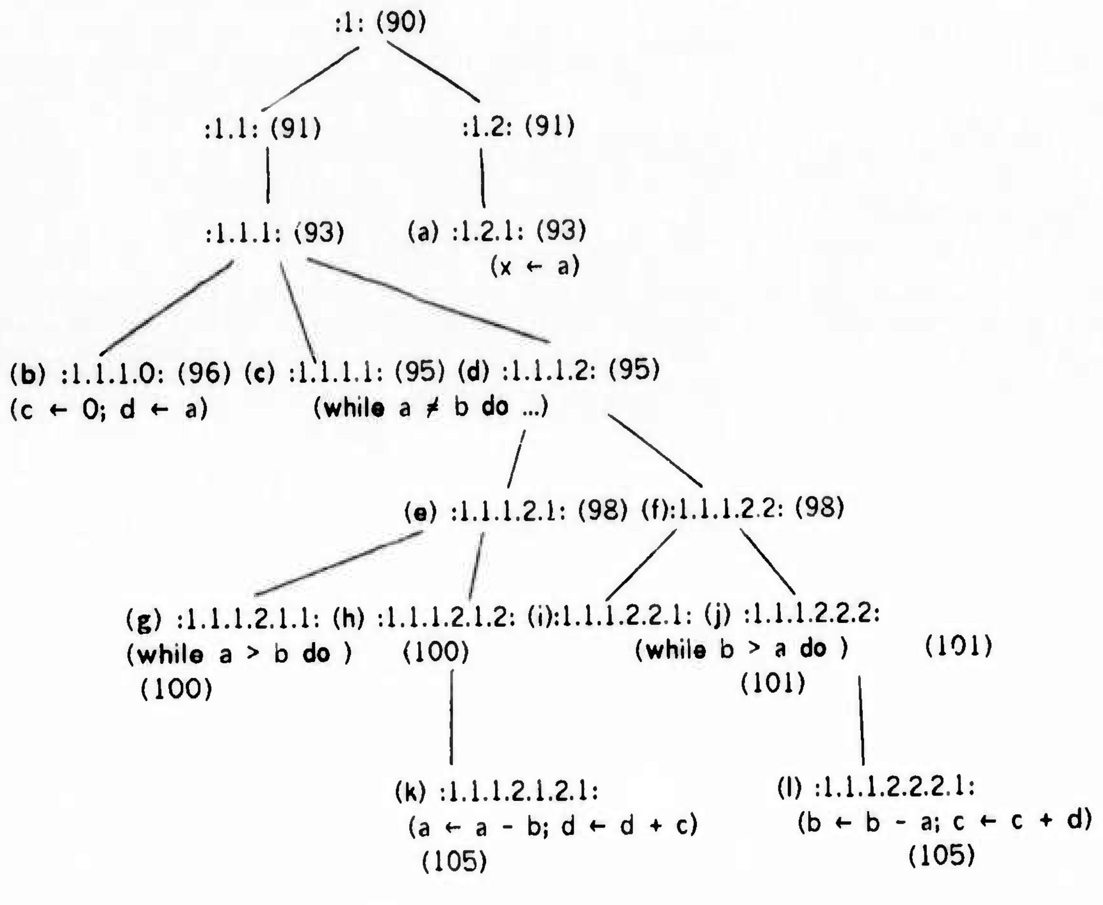
(C): ((**a**) ((**g** , **h**) ((**b**) (**c** , **e**)) 1.01) 1.01) .451

(A) corresponds to a decomposition which is based on the best decomposition at the last stage. Both (B) and (C) are better decompositions for the program representation from which several assumptions are hidden. (B) corresponds to the situation in version I where control mechanisms appear together. Here **c** and **e** represent the two inner **while** constructions. (C) indicates that the objects which assign values to x, a and b interact with the program less than the control mechanisms.

A final version of the progran can be elaborated, in much the same way as above, but where even more assumptions are made by the objects. Specifically:

Compute the value of the greatest common divisor of the initial(positive) contents of the integer variables a and b, and leave the result in the variable x; also leave in the variable y the value of the least common multiple of a and b, i.e. a*b / gcd(a,b), but without using either multiplication or division.

A GCD Computation (Version III)


:1: (90)
:1.1: (91)
:1.2: (91)
:1.1.1: (93)
(a) :1.2.1: (93) (x ← a)
(b) :1.1.1.0: (96) (c ← 0; d ← a)
(c) :1.1.1.1: (95) (while a ≠ b do ...)
(d) :1.1.1.2: (95)
(e) :1.1.1.2.1: (98)
(f):1.1.1.2.2: (98)
(g) :1.1.1.2.1.1: (while a > b do) (100)
(h) :1.1.1.2.1.2: (100)
(i):1.1.1.2.2.1: (101)
(j) :1.1.1.2.2.2: (while b > a do) (101)
(k) :1.1.1.2.1.2.1: (a ← a - b; d ← d + c) (105)
(l) :1.1.1.2.2.2.1: (b ← b - a; c ← c + d) (105)

This development is essentially identical with version II except that the least common multiple (lcm) of the initial contents of a and b is computed in addition to the gcd of these values.

:1: states the problem. :1.1: has the effect of leaving c + d = A * B / gcd(A,B) and a = gcd(A,B).

:1.2: assigns x the value of gcd(A,B) and y the value of lcm(A,B). The relationships c + d = lcm(A,B) and a = gcd(A,B) are hidden from :1.2.1:.

Object :1.1.1.0: was introduced to initialize c and d so that the invariant A * B = a * c + b * d holds, prior to the execution of :1.1.1.1:.

The remainder of the tree is the same as the corresponding objects in version II except that the invariant A * B = a * c + b * d is maintained in addition to gcd(A,B) = gcd(a,b).

:1:

assumptions: neither multiplication nor division is to be used and a > 0, b > 0, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, write access to x is required, write access to y is required, neither multiplication nor division is to be used :1: COMPUTE THE GCD AND THE LCM OF a AND b, LEAVE THE GCD IN THE VARIABLE x AND THE LCM IN THE VARIABLE y.

effects and
post-condition: x = gcd(A,B) ∧ y = lcm(A,B)

verification: Since the gcd and lcm of any two positive integers are always defined and are computable, the computation is feasible.

The object/assumption table for :1: is

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
:1:	1	1	1	1	1	1					1	1														1

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
25) write access to y is required

We elaborate :1: as

:1.1:

assumptions: neither multiplication nor division is to be used and $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, write access to a is required, write access is required for c, write access is required for d, c and d are integer variables, neither multiplication nor division is to be used, $(a=b \wedge gcd(A,B) = gcd(a,b) \wedge a*c + b*d = A*B) \supset a = gcd(A,B) \wedge c + d = lcm(A,B)$

:1.1: REPLACE THE VALUE OF a BY gcd(A,B) AND LEAVE THE EXPRESSION c + d EQUAL TO lcm(A,B)

effects and **post-condition:** $a = gcd(A,B) \wedge c + d = lcm(A,B)$

:1.2:

assumptions: $a = gcd(A,B)$, write access to x is required, read access to a is required, c and d are integer variables , $c + d = lcm(A,B)$, write access to y is required, read access is required for both c and d, neither multiplication nor division is to be used

:1.2: REPLACE THE VALUE OF x BY THE VALUE CONTAINED IN a AND THE VALUE OF y BY c + d.

effects: $x = a \wedge y = lcm(A,B)$

post-condition: $x = gcd(A,B)$

verification: In order for $x = gcd(A,B)$ and $y = lcm(A,B)$ to hold after :1.2:, given that its effect is $x = a$ and $y = c + d$, its assumptions must be $a = gcd(A,B)$ and $c + d = lcm(A,B)$. But this is guaranteed as the post-condition of :1.1:. Further, :1.1: requires the same assumptions as :1: plus the ability to store into a.

USING THE MEASURE
A GCD COMPUTATION

object/assumption table for :1.1:, :1.2:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
:1.1:	I	I	I		I	I		I			I	I															I			I	I
:1.2:	I			I		I	I																			I		I	I		

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) $a = gcd(A,B)$
7) $(a=b \wedge gcd(a,b)=gcd(A,B) \wedge a*c+b*d=A*B) \supset$
$(a=gcd(A,B) \wedge c+d=lcm(A,B))$
10) write access to b is required
11) read access to b is required
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
28) $c + d = lcm(A,B)$
29) write access is required for c
30) write access is required for d

Next an elaboration of :1.1: can be made by hiding

$$(a=b \wedge gcd(a,b)=gcd(A,B) \wedge a*c+b*d=A*B) \supset$$

$$(a=gcd(A,B) \wedge c+d=lcm(A,B))$$

(This has already appeared as an assumption for :1.1:.)

:1.1.1:

assumptions: neither multiplication nor division is to be used and a > 0, b > 0, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read and write access is required for both a and b, c and d are integer variables , read access is required for both c and d, write access is required for c, write access is required for d, neither multiplication nor division is to be used

:1.1.1: MAKE a = b SUCH THAT gcd(a,b) = gcd(A,B) AND c AND d SUCH THAT a*c + b*d = A*B.

effects and
post-condition: $a = b, gcd(a,b) = gcd(A,B) \wedge a*c + b*d = A*B$

An elaboration of :1.2: can be made if we observe that

$$(a = gcd(A,B) \wedge c+d = lcm(A,B)) \supset \textbf{true}$$

and

$$(x = a \wedge y = c + d) \supset (x = gcd(A,B) \wedge y = lcm(A,B))$$

But since :1.2: assumes a = gcd(A,B) we can write

(a) :1.2.1:

assumptions: write access to x is required, read access to a is required, write access to y is required, c and d are integer variables , read access is required for both c and d, neither multiplication nor division is to be used

:1.2.1: $x \leftarrow a$; $y \leftarrow c + d$;

effect and
post-condition: $x = a, y = c + d$

As a result, the object/assumption table for :1.1, :1.2:, :1.1.1:, and :1.2.1: is

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
:1.1:	|	|	|		|	|					|	|															|			|	|
:1.2:	|			|		|	|		◆	◆																|		|	|		
:1.1.1:	|	|	|		|	|					|	|															|			|	|
(a) :1.2.1:	|			|		|																				|		|			

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) a = gcd(A,B)
8) (a = gcd(A,B) ∧ c+d = lcm(A,B)) ⊃ **true**
9) (x = a ∧ y = c + d) ⊃ (x = gcd(A,B) ∧ y = lcm(A,B))
10) write access to b is required
11) read access to b is required
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
28) c + d = lcm(A,B)
29) write access is required for c
30) write access is required for d

Adopting the convention that a' and b' equal the values of a and b just prior to the most recent execution of :1.1.1.2: we elaborate :1.1.1:

(c) :1.1.1.1:

assumptions: neither multiplication nor division is to be used and $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $[((a < a' \vee b < b') \wedge gcd(a,b) = gcd(A,B) \wedge a \neq b) \supset (a \neq b \wedge gcd(a,b) = gcd(a',b')) \wedge max(a',b') > max(a,b))]$, c and d are integer variables, $a*c + b*d = A*B$, neither multiplication nor division is to be used

:1.1.1.1: **while** $a \neq b$ **do** :1.1.1.2: ;

post-condition: $a = b$, $gcd(a,b) = gcd(A,B)$, $c + d = lcm(A,B)$

(d) :1.1.1.2:

assumptions: neither multiplication nor division is to be used and $a > 0$, $b > 0$, A symbolizes the initial value of a, B symbolizes the initial value of b, a is an integer variable, b is an integer variable, read and write access is required for both a and b, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $a \neq b$, $gcd(a,b) = gcd(A,B)$, write access is required for c, write access is required for d, read access is required for both c and d, c and d are integer variables , $a*c + b*d = A*B$, neither multiplication nor division is to be used

:1.1.1.2: DECREASE EITHER a, b, OR BOTH a AND b SUCH THAT gcd(a,b) = gcd(A,B) AND INCREASE c, d OR BOTH SUCH THAT a*c + b*d = A*B.

effect and
post-condition: $(a < a' \vee b < b')$, $gcd(a,b) = gcd(A,B)$

Note that the condition a*c + b*d = A*B cannot be guaranteed since no values have been assigned to c or to d and since no previous part of the program has guaranteed this condition. Hence, :1.1.1.0: is introduced as

(b) :1.1.1.0:

assumptions: c and d are integer variables, write access is required for c, write access is required for d, A = a ∧ B = b, read access to a is required, read access to b is required, neither multiplication nor division is to be used and a > 0, b > 0, a and b are integer variables, A and B symbolize the respective initial values of a and b, neither multipliction nor division is to be used

:1.1.1.0: ASSIGN VALUES TO c AND d SUCH THAT a*c ; b*d = A*B, i.e. c ← 0; d ← a;

effects: c = 0 ∧ d = a

post-condition: a*c + b*d = A*B

verification: Since A = a and B = b and c = 0 and d = a then a * c + b * d = a * b.

Here the statement of :1.1.1.0: also includes its own elaboration. Most of the assumptions for :1.1.1.0: could be hidden from an elaboration which would read "ASSIGN ZERO TO c AND THE VALUE OF a TO d".

The object/assumption table is then:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(b)	:1.1.1.0:	1	1	1			1						1															1			1	1		1
(c)	:1.1.1.1:	1	1	1		1							1	1	1	1												1					1	
(d)	:1.1.1.2:	1	1	1		1	1					1	1	1		1	1											1	1		1	1	1	

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
B symbolizes the initial value in b
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
prior to the last execution of :1.1.1.2:
13) $(((a<a' \vee b<b') \wedge gcd(a,b)=gcd(A,B) \wedge a \neq b) \supset$
$(a \neq b \wedge gcd(a,b) = gcd(A,B)) \wedge max(a',b') > max(a,b))$
14) $gcd(a,b) = gcd(A,B)$
15) $a \neq b$
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
29) write access is required for c
30) write access is required for d
31) $a*c + b*d = A*B$
32) $A = a \wedge B = b$

Because of the large number of assumptions made by **a, b, c,** and **d** all possible decompositions have the same entropy loading values. This is an instance of **saturation in a table**. The decomposition which was best for the previous developments of this program at this stage is

((**a**) ((b) (**c** , **d**)) 1.39) 1.39

We add the following assumption to :1.1.1.2:

$$gcd(a',b') = gcd(a,b) \equiv gcd(a,b) = gcd(A,B)$$

(e) :1.1.1.2.1:

assumptions:

$a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b write access to a is required, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $gcd(a,b) = gcd(a',b')$, c and d are integer variables , read access is required for both c and d, write access is required for c, write access is required for d, $a*c + b*d = A*B$, neither multiplication nor division is to be used

:1.1.1.2.1: IF POSSIBLE, MAKE a SMALLER THAN b SUCH THAT gcd(a,b) = gcd(a',b') AND MAKE c AND d SUCH THAT a*c + b*d = A*B.

effects and post-condition:

$a \leq a' \wedge gcd(a,b) = gcd(a',b') \wedge a'>b' \supset b'\geq a \wedge a*c + b*d = A*B$

(f) :1.1.1.2.2:

assumptions:

$a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b read access to a is required, write access to b is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, $gcd(a,b) = gcd(a',b')$, c and d are integer variables , read access is required for both c and d, write access is required for c, write access is required for d, $a*c + b*d = A*B$, neither multiplication nor division is to be used

:1.1.1.2.2: IF POSSIBLE, MAKE b SMALLER THAN a SUCH THAT gcd(a,b)=gcd(a',b') AND MAKE c AND d SUCH THAT a*c +b*d = A*B.

effects and post-condition:

$gcd(a,b) = gcd(a',b') \wedge b\leq b' \wedge (b' > a \supset a \geq b)$

verification:

To show that the post-condition for :1.1.1.2: holds, i.e. $(a < a' \vee b < b') \wedge gcd(a,b) = gcd(a',b')$ a*c

+ b*d = A*B, holds. Case 1: if a' > b' then b ≥ a and a < a', since after the execution of :1.1.1.2.1.2:, gcd(a,b) = gcd(a',b').

Case 2: If b' > a then a ≥ b which means that b' > b. But since gcd(a,b) = gcd(a',b') the post-condition holds. For both cases, a*c + b*d = A*B also holds.

The relevant table is then

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
(e)	:1·1·1·2·1:	1	1	1		1	1						1	1					1									1	1		1	1	1
(f)	:1·1·1·2·2:	1	1	1			1					1	1	1					1									1	1		1		1

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
29) write access is required for c
30) write access is required for d
31) a*c + b*d = A*B
32) A = a ∧ B = b

RLB is ((a) ((b) (c , d)) .950) .950

RUB is ((a) ((b) (c , d)) 1.33) 1.33

Unfortunately, the actual values are

((a) ((b) ((c) (e , f)) 1.61) 1.61) 1.33

This is also a best decomposition. One reason for this increase in entropy loadings is because more assumptions are shared among the objects.

We elaborate :1.1.1.2.1: as

(g) :1.1.1.2.1.1:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') $\wedge$ a > b $\supset$ gcd(a',b') = gcd(a-b,b), c and d are integer variables , a*c + b*d = A*B, neither multiplication nor division is to be used

:1.1.1.2.1.1: **while** a > b **do**

(h) :1.1.1.2.1.2:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b write access to a is required, read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') = gcd(a-b,b) $\wedge$ A*B = (a-b)*c + b*(d+c), c and d are integer variables , a*c + b*d = A*B, read access is required for both c and d, write access is required for d, neither multiplication nor division is to be used

:1.1.1.2.1.2: DECREASE a BY b AND INCREASE d BY c.

effects and **post-conditions:** a has been decreased by b and d has been increased by c

effect and **post-contition:** $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, gcd(a',b') = gcd(a,b), a*c + b*d = A*B

Similarly, we elaborate :1.1.1.2.2:

(i) :1.1.1.2.2.1:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b read access to a is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), (gcd(a,b') $\wedge$ b > a) $\supset$ gcd(a',b') =gcd(a,b-a), c and d are integer variables , a*c + b*d = A*B, neither multiplication nor division is to be used

:1.1.1.2.2.1: **while** b > a **do**

(j) :1.1.1.2.2.2:

assumptions: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, A symbolizes the initial value in a and B symbolizes the initial value in b read access to a is required, write access to b is required, read access to b is required, a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:, gcd(a,b) = gcd(a',b'), gcd(a',b') = gcd(a,b-a) $\wedge$ A*B = a*(c+d) + (b-a)*d, c and d are integer variables, a*c + b*d = A*B, read access is required for both c and d, write access is required for c, neither multiplication nor division is to be used

:1.1.1.2.2.2: DECREASE b BY a AND INCREASE c BY d;

effects and
post-conditions: b has been decreased by a and c has been increased by d

effects and
post-condition: $a > 0 \wedge b > 0 \wedge$ a and b are integer variables, gcd(a,b) = gcd(a',b') $\wedge$ a*c + b*d = A*B

The table for these two parts is then

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
(g)	:1·1·1·2·1·1:	1	1	1			1						1	1					1						1			1					1
(h)	:1·1·1·2·1·2:	1	1	1		1	1						1	1					1		1							1	1			1	1
(i)	:1·1·1·2·2·1:	1	1	1			1						1	1					1							1		1					1
(j)	:1·1·1·2·2·2:	1	1	1			1					1	1	1					1			1						1	1		1		1

0) neither multiplication nor division is to be used
1) a > 0, b > 0, a and b are integer variables
2) A symbolizes the initial value in a and
   B symbolizes the initial value in b
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
    prior to the last execution of :1.1.1.2:
17) gcd(a,b) = gcd(a',b')
19) gcd(a',b') = gcd(a-b,b) ∧ A∗B = (a-b)∗c + b∗(d+c)
20) gcd(a',b') = gcd(a,b-a) ∧ A∗B = a∗(c+d) + (b-a)∗d
23) (gcd(a',b') ∧ a > b ∧ a∗c + b∗d = A∗B) ⊃ gcd(a',b') = gcd(a-b,b) ∧ A∗B = (a-b)∗c + b∗(d+c))
24) (gcd(a',b') ∧ b > a ∧ a∗c + b∗d = A∗B) ⊃ gcd(a',b') = gcd(a, b-a) ∧ a∗c + b∗d = A∗B
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
28) c + d = lcm(A,B)
29) write access is required for c
30) write access is required for d
31) a∗c + b∗d = A∗B

RLB and RUB from

((a) ((b) ((c) (e , f))))

for the above expansion are

RLB: ((a) ((b) ((c) (e , f))) .796) .796

RUB: ((a) ((b) ((c) (e , f)) 1.55) 1.55) 1.54

The actual entropy loading values are

((a) ((b) ((c) (g , h , i , j)) 1.48) 1.75) 1.28

which is again a best decomposition. If (g , h , i, j) is decomposed, the best decomposition is

((g , i) (h , j)) 1.55

which indicates that the bodies of the two loops interact less with the control mechanisms for those loops than do the loops with each other.

Lastly, the assignments can be elaborated if we add assumptions to the table as follows

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
(k)	:1·1·1·2·1·2:	I	I	I		I	I						I	I					I		I		◆					I	I			I	I
(l)	:1·1·1·2·2·2:	I	I	I			I					I	I	I					I			I		◆				I	I		I		I

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
4) write access to a is required
5) read access to a is required
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b prior to the last execution of :1.1.1.2:
17) $gcd(a,b) = gcd(a',b')$
19) $gcd(a',b') = gcd(a-b,b)$
20) $gcd(a',b') = gcd(a,b-a)$
21) $(gcd(a',b') = gcd(a,b-a) \wedge A*B = (a-b)*c + b*(d+c)) \supset$ **true**,
b decreased by a $\supset gcd(a',b') = gcd(a,b)$
22) $(gcd(a',b') = gcd(a-b,b) \wedge A*B = a*(c+d) + (b-a)*d) \supset$ **true**,
a decreased by b $\supset gcd(a,b) = gcd(a',b')$
23) $(gcd(a',b') \wedge a > b \wedge a*c + b*d = A*B) \supset gcd(a',b') = gcd(a-b,b) \wedge$
$A*B = (a-b)*c + b*(d+c))$
24) $(gcd(a',b') \wedge b > a \wedge a*c + b*d = A*B) \supset gcd(a',b') = gcd(a, b-a) \wedge$
$a*c + b*d = A*B$
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
28) $c + d = lcm(A,B)$
29) write access is required for c
30) write access is required for d
31) $a*c + b*d = A*B$

(k) :1.1.1.2.1.2.1:

assumptions: write access to a is required, read access to a is required, read access to b is required read access is required for both c and d, write access is required for d, neither multiplication nor division is to be used

:1.1.1.2.1.2.1: a ← a - b; d ← d + c;

effects and **post-condition:** a decreased by b and d increased by c.

and

(l) :1.1.1.2.2.2.1:

assumptions: read access to a is required, write access to b is required, read access to b is required read access is required for both c and d, write access is required for c, neither multiplication nor division is to be used

:1.1.1.2.2.2.1: b ← b - a; c ← c + d;

effects and **post-condition:** b decreased by a and c increased by d

Now we can display the table for this new development.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
	:1:	│	│	│	│	│	│		│			│	│														│							
	:1·1:	│	│	│		│	│					│	│															│			│	│		
	:1·2:	│			│		│	│		♦	♦																│		│	│				
	:1·1·1:	│	│	│		│	│					│	│															│			│	│		
(a)	:1·2·1:	│			│		│																				│		│					
(b)	:1·1·1·0:	│	│	│			│						│															│			│	│		│
(c)	:1·1·1·1:	│	│	│		│							│	│	│													│					│	
(d)	:1·1·1·2:	│	│	│		│	│					│	│	│		│	│											│	│		│	│	│	
(e)	:1·1·1·2·1:	│	│	│		│	│						│	│					│									│	│		│	│	│	
(f)	:1·1·1·2·2:	│	│	│			│					│	│	│					│									│	│		│		│	
(g)	:1·1·1·2·1·1:	│	│	│			│						│	│					│						│			│					│	
(h)	:1·1·1·2·1·2:	│	│	│		│	│						│	│					│		│		♦					│	│			│	│	
(i)	:1·1·1·2·2·1:	│	│	│			│						│	│					│							│		│					│	
(j)	:1·1·1·2·2·2:	│	│	│			│					│	│	│					│			│		♦				│	│		│		│	
(k)	:1·1·1·2·1·2·1:	│	│			│	│						│															│	│			│		
(l)	:1·1·1·2·2·2·1:	│	│				│					│	│															│	│		│			

0) neither multiplication nor division is to be used
1) $a > 0$, $b > 0$, a and b are integer variables
2) A symbolizes the initial value in a and
B symbolizes the initial value in b
3) write access to x is required
4) write access to a is required
5) read access to a is required
6) $a = gcd(A,B)$
7) $(a=b \wedge gcd(a,b)=gcd(A,B) \wedge a*c+b*d=A*B) \supset$
$(a=gcd(A,B) \wedge c+d=lcm(A,B))$
8) $(a = gcd(A,B) \wedge c+d = lcm(A,B)) \supset$ **true**
9) $(x = a \wedge y = c + d) \supset (x = gcd(A,B) \wedge y = lcm(A,B))$
10) write access to b is required
11) read access to b is required
12) a' and b' equal the respective values of a and b
prior to the last execution of :1.1.1.2:
13) $(((a<a' \vee b<b') \wedge gcd(a,b)=gcd(A,B) \wedge a \neq b) \supset$
$(a \neq b \wedge gcd(a,b) = gcd(A,B)) \wedge max(a',b') > max(a,b))$
14) $gcd(a,b) = gcd(A,B)$
15) $a \neq b$
16) $gcd(a,b) = gcd(A,B) \equiv gcd(a',b') = gcd(a,b)$
17) $gcd(a,b) = gcd(a',b')$
18) $(a > b \wedge gcd(a',b') = gcd(a-b,b) \vee$
$a \leq b \wedge gcd(a',b') = gcd(a,b-a)$
19) $gcd(a',b') = gcd(a-b,b) \wedge A*B = (a-b)*c + b*(d+c)$
20) $gcd(a',b') = gcd(a,b-a) \wedge A*B = a*(c+d) + (b-a)*d$
21) $(gcd(a',b') = gcd(a,b-a) \wedge A*B = (a-b)*c + b*(d+c)\ ) \supset$ **true,**
b decreased by a $\supset gcd(a',b') = gcd(a,b)$
22) $(gcd(a',b') = gcd(a-b,b) \wedge A*B = a*(c+d) + (b-a)*d) \supset$ **true,**
a decreased by b $\supset gcd(a,b) = gcd(a',b')$

23) $(gcd(a',b') \wedge a > b \wedge a*c + b*d = A*B) \supset gcd(a',b') = gcd(a-b,b) \wedge A*B = (a-b)*c + b*(d+c))$
24) $(gcd(a',b') \wedge b > a \wedge a*c + b*d = A*B) \supset gcd(a',b') = gcd(a, b-a) \wedge a*c + b*d = A*B$
25) write access to y is required
26) c and d are integer variables
27) read access is required for both c and d
28) $c + d = lcm(A,B)$
29) write access is required for c
30) write access is required for d
31) $a*c + b*d = A*B$
32) $A = a \wedge B = b$

Even though information has been hidden from (k) (l), entropy loadings remain the same as for the previous decompositions for

(g , i) (h , j),

but increase for ((c) ((g , i) (k , l))), i.e.

((a) ((b) ((c) ((g , i) (k , l)) 1.55) 1.75) 1.75) 1.28

Because little information was hidden from the objects at early stages, entropy loadings tended to be larger than in versions I and II.

* * *

The three versions of this gcd computation have indicated that control mechanisms usually share more assumptions in a program than the objects whose execution is being controlled. These examples also illustrated several instances where information was hidden from objects. In versions I and II, this resulted in entropy loadings that were smaller than corresponding loadings had assumptions not been hidden. Version III, however, indicated that more information was shared than in versions I and II. Entropy loading figures can be improved if the texts which compute the gcd are separated from those which compute the lcm.

A SEQUENCES PROBLEM

This example is also due to Dijkstra[DJ3 pp.53-63], but makes use of some of the notational conventions due to Hoare[HO3]. The specific conventions are:

a) <empty> is a **sequence**
b) if x is a sequence and d can be an element of a sequence then
$x^\frown d$ is a sequence

c) The only sequences are those defined by (a) and (b)
d) $(x^\frown d)$.last = d
e) initial$(x^\frown d)$ = x
f) $x^\frown(y^\frown z) = (x^\frown y)^\frown z$
g) d.first = d
h) x $\neq$ <empty> $\supset$ $(x^\frown d)$.first = x.first
i) final(d) = <empty>
j) x $\neq$ <empty> $\supset$ final$(x^\frown d)$ = final$(x)^\frown d$

last, initial, first, and **final** are
not defined for <empty>.

k) **length**(<empty>) = 0
l) **length**$(x^\frown d)$ = succ(length(x))
m) x:$^\frown$d means x ← $x^\frown d$
n) d **from** x **means** d ← x.first; x ← final(x)
o) d **back from** x means d ← x.last; x ← initial(x)
p) **from** x means x ← final(x)
q) **back from** x means x ← initial(x)

Consider the sequences constructed from the digits 1, 2, and 3 which contain no occurrence of two adjacent, identical subsequences. Call these sequences "good". Several examples of good sequences are

1
21
1312
31213

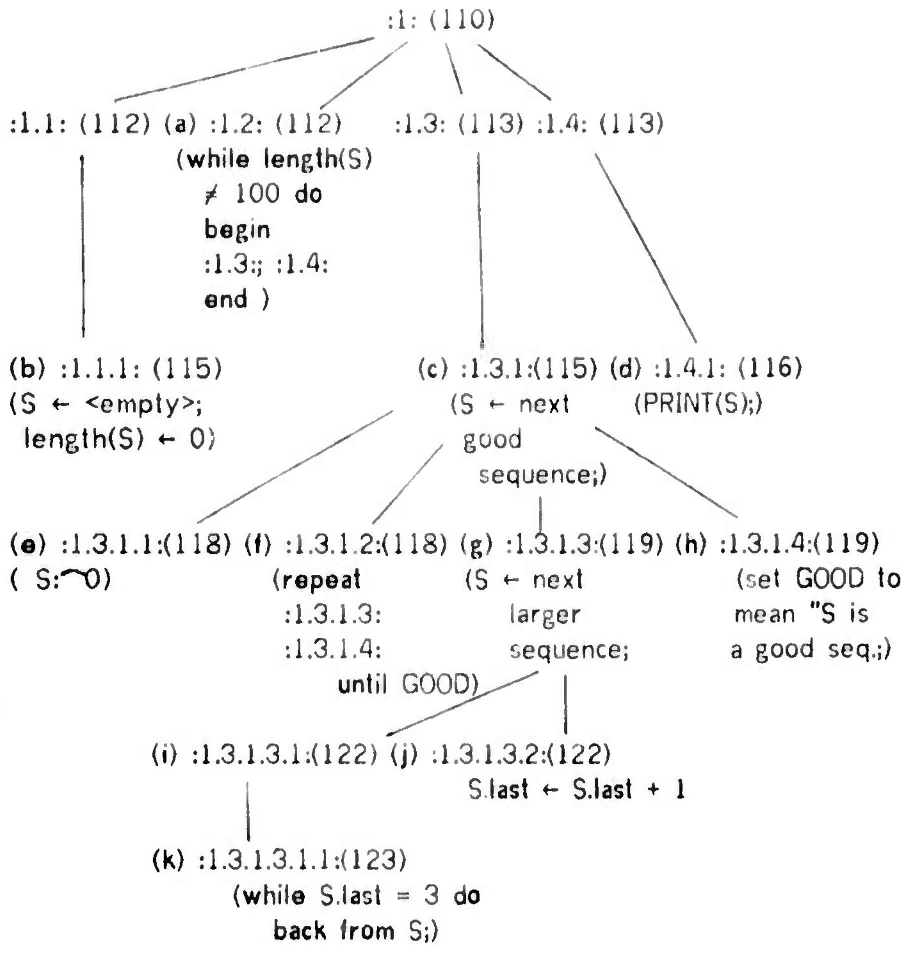
Several sequences which are not "good" are

22
123123
321232123

The problem can now be stated:

Assuming that there exists a good sequence of length 100, write a program which generates the list of good sequences in lexicographic order up to and including the first good sequence of length 100. (Here, 1 precedes 2 which precedes 3).

A SEQUENCES PROBLEM



```
:1: (110)

:1.1: (112)   (a) :1.2: (112)        :1.3: (113)   :1.4: (113)
                  (while length(S)
                     ≠ 100 do
                     begin
                     :1.3:; :1.4:
                     end )

(b) :1.1.1: (115)          (c) :1.3.1:(115)   (d) :1.4.1: (116)
(S ← <empty>;                  (S ← next          (PRINT(S);)
 length(S) ← 0)                 good
                                sequence;)

(e) :1.3.1.1:(118)  (f) :1.3.1.2:(118)  (g) :1.3.1.3:(119)  (h) :1.3.1.4:(119)
( S:⌒0)             (repeat              (S ← next           (set GOOD to
                     :1.3.1.3:            larger              mean "S is
                     :1.3.1.4:            sequence;           a good seq.;)
                          until GOOD)

(i) :1.3.1.3.1:(122)  (j) :1.3.1.3.2:(122)
                          S.last ← S.last + 1

(k) :1.3.1.3.1.1:(123)
      (while S.last = 3 do
         back from S;)
```

(l) :2.1:(130)
length(S) is length

(m) :2.2:(130)
S.last is d[length]

(n) :2.3:(130)
write access to S.last is d[length] ← ...

(o) :2.4: S ← S⌒... (131)
length ← length + 1;
d[length] ← ...

(p) :2.5: (131)
back from S
length ← length - 1

(q) :2.6: (131)
S ← <empty>
length ← 0

(r) :2.7: read access to elements of S. d[1] ... d[length]
(132)

(s) :2.8: write access to S
d[1] ← ..., ... d[length] ← ...
(132)

This problem requires that a list of lexicographically ordered sequences of 1's, 2's, and 3's, containing no adjacent identical subsequences, be printed. This list should terminate with the first sequence whose length is 100.

Object :1.1: sets S to <empty>. Objects :1.3: and :1.4: generate and print the next good sequence. Object :1.2: controls objects :1.3: and :1.4: until a sequence of length 100 is produced.

Object :1.3.1.1: extends S with 0. Objects :1.3.1.3: and :1.3.1.4: produce the next lexicographically larger sequence and test whether it is a good sequence. :1.3.1.3: and :1.3.1.4: are controlled by :1.3.1.2: until a good sequence is found.

Objects :1.3.1.3.1.1: and :1.3.1.3.2: remove trailing 3's from S and increment the last element of S by 1.

Objects :2.1: through :2.8: implement the operations required by the objects to manipulate a sequence. S is implemented in terms of an array and several simple variables.

:1:

assumptions: OS = <empty>, INIT(R,S) is defined to be {if length(R) = 0 then true else R.first = S.first ∧ INIT(final(R), final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, there exists a good sequence of length 100

:1: OUTPUT THE INITIAL PORTION OF THE SEQUENCE OF LEXICOGRAPHICALLY ORDERED SEQUENCES OF 1's, 2's,AND 3's, SUCH THAT NO SEQUENCE CONTAINS TWO ADJACENT IDENTICAL SUBSEQUENCES. TERMINATE THE LIST WITH THE FIRST SUCH SEQUENCE WHOSE LENGTH EQUALS 100.

effects and post-conditions: INIT(OS,Q) ∧ length(OS.last) = 100

verification: If there exists a "good" sequence of length 100 and the effects of :1: agree with the post-condition, the post-condition is satisfied.

The object/assumption table for :1: is

	1	2	3
:1:	1	1	1

1) $\mathcal{O}$S = <empty>
2) INIT(R,S) is defined to be {**if** length(R) = 0 **then true**
else R.first = S.first $\wedge$ INIT(final(R),final(S))}
$\wedge$ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
$\wedge$ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100

We elaborate :1: as follows, with all the pre-conditions derived
:1.1:

assumptions: requires ability to set S to <empty> and **length**(S) to 0, OS = <empty>, INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.first = S.first $\wedge$ INIT(final(R), final(S))} $\wedge$ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" $\wedge$ Q represents the sequence of lexicographically ordered "good" sequences, there exists a good sequence of length 100

:1.1: SET SEQUENCE S TO <empty> AND **length**(S) TO ZERO.

effects and post-conditions: S = <empty>, **length**(S) = 0, INIT(OS,Q), **length**(OS.last) = 100

(a) :1.2:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.first = S.first $\wedge$ INIT(final(R),final(S))} $\wedge$ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" $\wedge$ Q represents the sequence of lexicographically ordered "good" sequences, there exists a good sequence of length 100, INIT(OS,Q) $\wedge$ OS.last = S, INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q), (INIT(OS,Q) $\wedge$ OS.last = S $\wedge$ **length**(S) $\neq$ 100) $\supset$ INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q) requires ability to read **length**(S),

```
:1.2: while length(S) ≠ 100 do
    begin :1.3: ; :1.4: end;
```

effects and post-conditions: INIT(OS,Q), **length**(OS.last) = 100

verification: The pre-condition for the **while** construction holds since (1) we assume that there exists a good sequence of length equal to 100; (2) OS = <empty> initially; (3) and that the first good sequence satisfies the pre-condition.

:1.3:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires ability to write each element of S, requires ability to access each element of S, requires ability to concatenate onto S, i.e. S ← S⁀d, requires ability to delete from the end of S, i.e. **back from** S, INIT(OS,Q) ∧ OS.**last** = S ∧ OS.**last** = S, INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q)

:1.3: TRANSFORM S TO THE NEXT GOOD SEQUENCE AFTER ITS CURRENT CONTENTS

effects: S ← "next good sequence after S"

post-conditions: INIT(OS⁀S,Q)

:1.4:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, INIT(OS⁀S,Q), requires ability to access each element of S,

:1.4: PRINT(S);

effects: OS ← OS⁀S

post-conditions: INIT(OS,Q) ∧ OS.**last** = S

The object/assumption table for this part is

	1	2	3	4	5	6	7	8	9	10	11	12	13
:1:	1	1	1										
:1.1:	1	1	1	1									
:1.2:		1	1		1	1	1	1					
:1.3:		1			1			1		1	1	1	1
:1.4:		1							1	1			

1) OS = <empty>
2) INIT(R,S) is defined to be {**if length**(R) = 0 **then true**
else R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))}
∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
∧ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and **length**(S) to 0
5) INIT(OS,Q) ∧ OS.**last** = S ∧ OS.**last** = S
6) (INIT(OS,Q) ∧ OS.**last** = S ∧ **length**(S) ≠ 100) ⊃
INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
7) requires ability to read **length**(S)
8) INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
9) INIT(OS⌢S,Q)
10) requires ability to access each element of S
12) requires ability to concatenate onto S, i.e. S ← S⌢d
13) requires ability to delete from the end of S, i.e. **back from** S

We can hide considerable information from the refinement for :1.1: by adding

(Assumptions 1), 2), 3) ⊃ **true**) ∧ (S = <empty> ∧ **length**(S) = 0 ⊃ Assumptions 1), 2), 3))

Similarly, :1.4:(PRINT(S)) does not require INIT(OS⌢S,Q). Hence we add

(INIT(OS⌢S,Q) ⊃ **true**) ∧ (OS = OS'⌢S where OS' equals OS prior to :1.4:) ⊃ (INIT(OS,Q) ∧ OS.**last** = S) ∧ OS.**last** = S))

Further, for :1.3: (TRANSFORM S TO THE NEXT GOOD SEQUENCE AFTER ITS CURRENT CONTENTS.) we can add the assumption below. (In the remainder of this development, "illt is used as an abbreviation for the phrase "is lexicographically less than".)

(INIT(OS⁀"next good sequence after current sequence in S",Q) ∧ INIT(OS,Q) ∧ OS.last = S ⊃ S' equals S prior to executing :1.3: ∧ S is a good sequence) ∧ (∀x[S' illt x illt S ⊃ x is not a good sequence] ∧ S is a good sequence ∧ S ≠ S' ⊃ INIT(OS⁀S,Q))

The elaborations are

(b) :1.1.1:

assumptions: requires ability to set S to <empty> and **length**(S) to 0

:1.1.1: SET SEQUENCE S TO <empty> AND **length**(S) TO ZERO.

effects and post-conditions: S = <empty>, **length**(S) = 0

(c) :1.3.1:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true** else R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires ability to access each element of S, requires ability to write each element of S, requires ability to concatenate onto S, i.e. S ← S⁀d, requires ability to delete from the end of S, i.e. **back** from S, S' = S, prior to executing :1.3: ∧ S is a good sequence

:1.3.1: TRANSFORM S TO THE NEXT GOOD SEQUENCE AFTER ITS CURRENT CONTENTS

effects: S ← "next good sequence after S"

post-conditions: ∀x[S' illt x illt S ⊃ x is not a good sequence] ∧ S is a good sequence ∧ S ≠ S' ⊃

(d) :1.4.1:

assumptions: requires ability to access each element of S

:1.4.1: PRINT(S);

effects: OS ← OS⁀S

post-conditions: (OS = OS'⁀S where OS' equals OS prior to :1.4:)

The table

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	:1.1:	I	I	I	I										♦			
(a)	:1.2:		I	I		I	I	I	I									
	:1.3:		I			I			I		I	I	I	I		♦		
	:1.4:		I							I	I						♦	
(b)	:1.1.1:				I													
(c)	:1.3.1:		I								I	I	I	I				I
(d)	:1.4.1:										I							

1) OS = <empty>
2) INIT(R,S) is defined to be {if **length**(R) = 0 **then true**
else R.first = S.first ∧ INIT(**final**(R),**final**(S))}
∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
∧ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and **length**(S) to 0
5) INIT(OS,Q) ∧ OS.last = S
6) (INIT(OS,Q) ∧ OS.last = S ∧ **length**(S) ≠ 100) ⊃
INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
7) requires ability to read **length**(S)
8) INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
9) INIT(OS⌢S,Q)
10) requires ability to access each element of S
11) requires ability to write each element of S
12) requires ability to concatenate onto S, i.e. S ← S⌢d
13) requires ability to delete from the end of S, i.e. **back from** S
14) (Assumptions 1), 2), 3) ⊃ **true**) ∧
(S = <empty> ∧ **length**(S) = 0 ⊃ Assumptions 1), 2), 3))
15) (INIT(OS⌢"next good sequence after current sequence in S",Q)
∧ INIT(OS,Q) ∧ OS.last = S ⊃
S' equals S prior to executing :1.3: ∧ S is a good sequence)
∧ (∀x[S' illt x illt S ⊃ x is not a good sequence]
∧ S is a good sequence ∧ S ≠ S' ⊃
INIT(OS⌢S,Q))
16) (INIT(OS⌢S,Q) ⊃ **true**) ∧ (OS = OS'⌢S where OS' equals OS prior to :1.4:
⊃ (INIT(OS,Q) ∧ OS.last = S
∧ OS.last = S))
17) S' = S ∧ S is a good sequence

After hiding information from :1.1.1:, :1.3.1:, and :1.4.1:, the best decomposition is

((b) (a , c , d)) .562

where (b) sets S to <empty> and a, c, d compute good sequences and print, respectively.

Next we elaborate :1.3.1: to

(e) :1.3.1.1:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires ability to concatenate onto S, i.e. S ← S⁀d, S' = S ∧ S is a good sequence

:1.3.1.1: EXTEND S WITH ZERO;

effects: S = S'⁀0

post-conditions: initial(S) is a good sequence ∧ S.last = 0

(f) :1.3.1.2:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires read access to the boolean variable GOOD, S" = S, ∃x[S" illt x ∧ **length**(x) ≤ **length**(S") ∧ x is a good sequence] ∧ [(initial(S) is good) ∧ S.last = 0 ⊃ S is not good]

:1.3.1.2: **repeat** :1.3.1.3: ; :1.3.1.4: ; **until** GOOD;

post-conditions: GOOD ∧ there is no sequence s such that S" illt s illt S ∧ s is a good sequence

(g) :1.3.1.3:

assumptions: INIT(R,S) is defined to be {if length(R) = 0 **then** **true** **else** R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires ability to access each element of S, requires ability to write each element of S, requires ability to concatenate onto S, i.e. S ← S⁀d, requires ability to delete from the end of S, i.e. back **from** S, S" = S, ∃x[S" illt x **length**(x) ≤ **length**(S")]

:1.3.1.3: SET S TO BE THE NEXT LEXICOGRAPHICALLY LARGER SEQUENCE AFTER THE CONTENTS OF S AT THE START OF :1.3.1.3:, (I.E. S"), SUCH THAT **length**(S) ≤ length(S").

effects and post-conditions: S" illt S, length(S) ≤ length(S"), ¬∃x[S" illt x illt S], initial(S) is good, S.last ∈ {1,2,3}

(h) :1.3.1.4:

assumptions: INIT(R,S) is defined to be {if length(R) = 0 **then** **true** **else** R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, requires ability to access each element of S, initial(S) is a good sequence, requires write access to the boolean variable GOOD

:1.3.1.4: SET THE VARIABLE, GOOD, TO MEAN "S is a good sequence"

effects: GOOD = if "S is a good sequence" **then** **true** **else** **false**

post-conditions: GOOD = if "S is a good sequence" **then** **true** **else** **false**, S ≠ S'

verification: Note first that the following are theorems:

(a) If initial(S) is a good sequence, but S is not a good sequence, then no extension of S will be good.

(b) The only time S should be extended, in an effort to find lexicographically larger good sequences is when S itself is a good sequence.

Now, the only place in the above program where S is extended is at :1.3.1:, where S is guaranteed to be good. :1.3.3: guarantees that no relevant sequence is missed.

The object/assumption table now becomes:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(e)	:1.3.1.1:		1										1					1							
(f)	:1.3.1.2:		1																		1		1	1	
(g)	:1.3.1.3:		1								1	1	1	1										1	1
(h)	:1.3.1.4:		1								1								,			1			

1) OS = <empty>
2) INIT(R,S) is defined to be {if length(R) = 0 then true
   else R.first = S.first ∧ INIT(final(R),final(S))}
   ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
   ∧ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and length(S) to 0
5) INIT(OS,Q) ∧ OS.last = S
6) (INIT(OS,Q) ∧ OS.last = S ∧ length(S) ≠ 100) ⊃
   INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q)
7) requires ability to read length(S)
8) INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q)
9) INIT(OS⁀S,Q)
10) requires ability to access each element of S
11) requires ability to write each element of S
12) requires ability to concatenate onto S, i.e. S ← S⁀d
13) requires ability to delete from the end of S, i.e. back from S
14) (Assumptions 1), 2), 3) ⊃ true) ∧
    (S = <empty> ∧ length(S) = 0 ⊃ Assumptions 1), 2), 3))
15) INIT(OS⁀"next good sequence after current sequence in S",Q)
    ∧ INIT(OS,Q) ∧ OS.last = S ⊃
    S' equals S prior to executing :1.3: ∧ S is a good sequence
    ∧ ∀x[S' illt x illt S ⊃ x is not a good sequence]

∧ S is a good sequence ∧ S ≠ S' ⊃
INIT(OS⁀S,Q)

16) (INIT(OS⁀S,Q) ⊃ true) ∧
(OS = OS'⁀S where OS' equals OS prior to :1.4:
⊃ (INIT(OS,Q) ∧ OS.last = S
∧ OS.last = S))

17) S' = S ∧ S is a good sequence

18) initial(S) is a good sequence

19)

20) requires read access to the boolean variable GOOD

21) requires write access to the boolean variable GOOD

22) ∃x[S" illt x ∧ length(x) ≤ length(S") ∧
x is a good sequence] ∧ [(initial(S) is good) ∧ S.last = 0
⊃ S is not good]

23) S" = S

24) ∃x[S" illt x length(x) ≤ length(S")]

The expansion of (c) - :1.3.1.3: - implies

RLB: ((b) (a , c , d)) .053

RUB: ((b) (a , c , d)) .410

with the actual entropy loadings

((b) (a , e , f , g , h , d)) .410

The best decomposition of this elaboration is

((b) ((d) ((a) ((e) ((f) (g , h)) 1.28) 1.28) .956) .683) .410

Here,(b) sets S to <empty>,(d) prints S, (a) is the outer loop while construction and (e) extends S with a zero. In this example, the objects which calculate the next lexicographically larger sequence and which decide whether S is a good sequence interact most.

Consider next an expansion of :1.3.1.3:

(i):1.3.1.3.1:

assumptions: INIT(R,S) is defined to be {**if length**(R) = 0 **then true else** R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, S" = S, initial(S) is a good sequence ∧ 0 ≤ S.last ≤ 3, ability to read S.**last**, requires ability to delete from the end of S, i.e. back from S

:1.3.1.3.1: REMOVE TRAILING 3's FROM S

effects and
post-conditions: INIT(S,S"), initial(S) is a good sequence ∧ 0 ≤ S.last ≤ 3

(j):1.3.1.3.2:

assumptions: INIT(R,S) is defined to be {**if length**(R) = 0 **then true else** R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, ability to read S.**last**, ability to write S.**last**

:1.3.1.3.2: S.**last** ← S.**last** + 1

effects and
post-conditions: S" illtS, length(S) ≤ length(S"), ¬∃[S" illt x illt S], initial(S) is good, S.**last** ∈ {1,2,3}

The expansion of :1.3.1.3: suggests

RLB: ((b) ((d) ((a) ((e) (f , g , h) .562) .801) .563) .138

RUB: ((b) ((d) ((a) ((e) (f , g , h) 1.32) .900) .693) .377

and the actual entropy loadings are

((b) ((d) ((a) ((e) ((h) ((f)(i , j)

1.21) .662) .727) .900) .562) .377

Since the assumptions made by an object imply any subset of those assumptions, we can add an additional assumption to :1.3.1.3.1:(the object which removes trailing three's) and hide "initial(S") is good and 0 ≤ S.last ≤ 3" from its expansion :1.3.1.3.1.1:

(k):1.3.1.3.1.1:

assumptions: INIT(R,S) is defined to be {if **length**(R) = 0 **then true else** R.first = S.first ∧ INIT(final(R),final(S))} ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences" ∧ Q represents the sequence of lexicographically ordered "good" sequences, S" = S, ability to read S.last, requires ability to delete from the end of S, i.e. **back from** S

```
:1.3.1.3.1.1: while S.last = 3 do

back from S;
```

effects and
post-conditions: INIT(S,S") ∧ S.last ≠ 3

The entropy loadings of the previous decomposition are identical for the refinement including the elaboration for :1.3.1.3.1.1:. :1.3.1.4:(decide whether S is a good sequence) can be expanded by hiding most of the assumptions of :1.3.1.4:. However, the assumption "initial(S) is a good sequence" simplifies the calculations since only adjacent sequences, one of which contains S.last, need be considered.

The table for the program is now

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	:1:	I	I	I																									
	:1·1:	I	I	I	I										◆														
(a)	:1·2:		I	I		I	I	I	I																				
	:1·3:		I			I			I		I	I	I	I		◆													
	:1·4:		I							I	I						◆												
(b)	:1·1·1:				I																								
(c)	:1·3·1:		I								I	I	I	I				I											
(d)	:1·4·1:										I																		
(e)	:1·3·1·1:		I										I					I											
(f)	:1·3·1·2:		I																		I		I	I					
(g)	:1·3·1·3:		I								I	I	I	I										I	I				
(h)	:1·3·1·4:		I								I								I			I							
(i)	:1·3·1·3·1:		I											I										I		I		I	
(j)	:1·3·1·3·2:		I																									I	I
(k)	:1·3·1·3·1·1:		I											I										I				I	

1) OS = <empty>
2) INIT(R,S) is defined to be {if **length**(R) = 0 **then true**
 else R.first = S.first ∧ INIT(**final**(R),**final**(S))}
 ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
 ∧ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and **length**(S) to 0
5) INIT(OS,Q) ∧ OS.last = S
6) (INIT(OS,Q) ∧ OS.last = S ∧ **length**(S) ≠ 100) ⊃
 INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
7) requires ability to read **length**(S)
8) INIT(OS⌢"next lexicographically larger good sequence than the value in S",Q)
9) INIT(OS⌢S,Q)
10) requires ability to access each element of S
11) requires ability to write each element of S
12) requires ability to concatenate onto S, i.e. S ← S⌢d
13) requires ability to delete from the end of S, i.e. **back from** S
14) (Assumptions 1), 2), 3) ⊃ **true**) ∧
 (S = <empty> ∧ **length**(S) = 0 ⊃ Assumptions 1), 2), 3))
15) INIT(OS⌢"next good sequence after current sequence in S",Q)
 ∧ INIT(OS,Q) ∧ OS.last = S ⊃
 S' equals S prior to executing :1.3: ∧ S is a good sequence
 ∧ ∀x[S' illt x illt S ⊃ x is not a good sequence]
 ∧ S is a good sequence ∧ S ≠ S' ⊃
 INIT(OS⌢S,Q)
16) (INIT(OS⌢S,Q) ⊃ **true**) ∧

(OS = OS'⌢S where OS' equals OS prior to :1.4:
⊃ (INIT(OS,Q) ∧ OS.last = S
∧ OS.last = S))

17) S' = S ∧ S is a good sequence
18) initial(S) is a good sequence
19)
20) requires read access to the boolean variable GOOD
21) requires write access to the boolean variable GOOD
22) ∃x[S" illt x ∧ length(x) ≤ length(S") ∧
x is a good sequence] ∧ [(initial(S) is good) ∧ S.last = 0
⊃ S is not good]
23) S" = S
24) ∃x[S" illt x length(x) ≤ length(S")]
25) initial(S) is a good sequence ∧
0 ≤ S.last ≤ 3
26) (Assumptions 2, 13,23,25, 27) ⊃
(Assumptions 2, 13, 23, 27)
27) ability to read S.last
28) ability to write S.last

At about this stage in the development, Dijkstra introduces a data structure, namely an array d[1:100] to hold the digits of S. The variable, length, is introduced such that S.**last** = d[length]. By making this decision, all references to operations involving S or the operator **length** require additional assumptions. These assumptions are as follows:

"ability to access **length**(S)" becomes

length

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length

"ability to read S.**last**" becomes

d[length]

and assumes

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to read d[length]

"ability to write S.**last**" becomes

d[length] ← ...

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write d[length]

"ability to concatenate onto S" becomes

length ← length + 1; d[length] ← ...

and assumes

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write the variable length, ability to write d[length] "ability to delete from the end of S, i.e. **back from** S" becomes

length ← length - 1;

and assumes

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write the variable length

"ability to set S to <empty>" becomes

length ← 0;

and assumes

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to write the variable length,

"ability to read every element of S" requires

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to read every element of d

"ability to write every element of S" requires

d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write the variable length, ability to write every element of d

The object/assumption table for this introduction is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
:1:	I	I	I																																
:1·1:	I	I	I	I										◆															I		I				
:1·2:		I	I		I	I	I	I																					I	I					
:1·3:		I			I			I		I	I	I	I		◆														I	I	I	I		I	I
:1·4:		I							I	I						◆													I	I					I
:1·1·1:				I																									I		I				
:1·3·1:		I								I	I	I	I				I												I	I	I	I		I	I
:1·4·1:										I																			I	I					I
:1·3·1·1:		I										I					I												I	I	I	I			
:1·3·1·2:		I																		I		I	I												
:1·3·1·3:		I								I	I	I	I										I	I					I	I	I	I		I	I
:1·3·1·4:		I								I								I			I								I	I				I	
:1·3·1·3·1:		I											I										I		I		I		I	I	I		I		
:1·3·1·3·2:		I																									I	I	I	I		I	I		
:1·3·1·3·1·1:		I											I										I				I		I	I	I		I		

1) OS = <empty>
2) INIT(R,S) is defined to be {if **length**(R) = 0 **then true**
else R.**first** = S.**first** ∧ INIT(**final**(R),**final**(S))}
∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and 3's containing no adjacent identical subsequences"
∧ Q represents the sequence of lexicographically ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and **length**(S) to 0
5) INIT(OS,Q) ∧ OS.**last** = S
6) (INIT(OS,Q) ∧ OS.**last** = S ∧ **length**(S) ≠ 100) ⊃
INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q)
7) requires ability to read **length**(S)
8) INIT(OS⁀"next lexicographically larger good sequence than the value in S",Q)
9) INIT(OS⁀S,Q)
10) requires ability to access each element of S
11) requires ability to write each element of S
12) requires ability to concatenate onto S, i.e. S ← S⁀d
13) requires ability to delete from the end of S, i.e. **back from** S
14) (Assumptions 1), 2), 3) ⊃ **true**) ∧
(S = <empty> ∧ **length**(S) = 0 ⊃ Assumptions 1), 2), 3))
15) INIT(OS⁀"next good sequence after current sequence in S",Q)
∧ INIT(OS,Q) ∧ OS.**last** = S ⊃
S' equals S prior to executing :1.3: ∧ S is a good sequence
∧ ∀x[S' illt x illt S ⊃ x is not a good sequence]
∧ S is a good sequence ∧ S ≠ S' ⊃
INIT(OS⁀S,Q)
16) (INIT(OS⁀S,Q) ⊃ **true**) ∧
(OS = OS'⁀S where OS' equals OS prior to :1.4:
⊃ (INIT(OS,Q) ∧ OS.**last** = S

$\wedge$ OS.last = S))
17) S' = S $\wedge$ S is a good sequence
18) initial(S) is a good sequence
19)
20) requires read access to the boolean variable GOOD
21) requires write access to the boolean variable GOOD
22) $\exists$x[S" illt x $\wedge$ **length**(x) $\leq$ **length**(S") $\wedge$
x is a good sequence] $\wedge$ [(initial(S) is good) $\wedge$ S.last = 0
$\supset$ S is not good]
23) S" = S
24) $\exists$x[S" illt x **length**(x) $\leq$ **length**(S")]
25) initial(S) is a good sequence $\wedge$
0 $\leq$ S.last $\leq$ 3
26) (Assumptions 2, 13,23,25, 27) $\supset$
(Assumptions 2, 13, 23, 27)
27) ability to read S.last
28) ability to write S.last
29) d is an array which contains the digits of S,
one digit per element $\wedge$ the variable length
indexes the last element of S $\wedge$ the sequence is
empty when length = 0
30) ability to read the variable length
31) ability to write the variable length
32) ability to write d[length]
33) ability to read d[length]
34) ability to read every element of d
35) ability to write every element of d

It will be noted that the shared information between the parts has increased significantly by the decision to distribute all the information about the implementation of the sequence. In particular, the entropy loadings for the last decomposition are

((b) ((d) ((a) ((e) ((h) ((f) (k , j))

1.21) 1.21) 1.91) 1.38) 1.21) 1.21

All these values are greater than or equal to the corresponding values from the same decomposition of the previous table.

As an alternative to distributing additional assumptions throughout the program, objects can be created which provide the effects which are needed.

(l) :2.1:

assumptions:	d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length :2.1: PROVIDE THE ABILITY TO ACCESS THE LENGTH OF S, I.E. ACCESS THE VALUE OF length
effects:	**length**(S) = length
post-conditions:	**length**(S) equals the length of S according to the definition of **length**

(m) :2.2:

assumptions:	d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to read d[length] :2.2: PROVIDE THE ABILITY TO READ S.**last**, I.E. SET THE VALUE OF S.**last** TO d[length]
effects:	S.last = d[length]
post-conditions:	S.last equals the value of the last element of S according to the definition of S.last

(n) :2.3:

assumptions:	d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write d[length] :2.3: PROVIDE THE ABILITY TO WRITE S.**last** I.E. d[length] ← ...
effects:	S.last can be used as a name to cause a value to be stored into d[length]
post-conditions:	A value has been assigned to A.last

(o) :2.4:

assumptions: d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write the variable length, ability to write d[length]

:2.4: PROVIDE THE ABILITY TO CONCATENATE ONTO S, I.E. length ← length + 1; d[length] ← ...

effects and **post-conditions:** The definition for concatenation is satisfied

(p) :2.5:

assumptions: d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write the variable length

:2.5: PROVIDE THE ABILITY TO DELETE THE END OF S, I.E. **back from** S, I.E. length ← length -1;

effects and **post-conditions:** the definition for **back from** S is satisfied.

(q) :2.6:

assumptions: d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to write the variable length

:2.6: PROVIDE THE ABILITY TO SET S TO <empty> I.E. length ← 0

effects: length = 0

post-conditions: S = <empty>

(r) :2.7:

assumptions: d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to read every element of d

:2.7: PROVIDE THE ABILITY TO READ EVERY ELEMENT OF S I.E. READ ACCESS to d[1] ... d[length]

effects and
post-conditions: every element of S is readable

(s) :2.8:

assumptions: d is an array which contains the digits of S, one digit per element ∧ the variable length indexes the last element of S ∧ the sequence is empty when length = 0, ability to read the variable length, ability to write every element of d

:2.8: PROVIDE ABILITY TO WRITE EVERY ELEMENT OF S d[1] ← ..., ... , d[length] ← ...

effects and
post-conditions: every element of S has been made writable.

The table for the program with these additional parts is

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	:1:	1	1	1																																
	:1·1:	1	1	1	1										♦																					
(a)	:1·2:		1	1		1	1	1	1																											
	:1·3:		1			1			1		1	1	1	1		♦																				
	:1·4:		1							1	1						♦																			
(b)	:1·1·1:				1																															
	:1·3·1:		1								1	1	1	1				1																		
(d)	:1·4·1:										1																									
(e)	:1·3·1·1:		1										1					1																		
(f)	:1·3·1·2:		1																		1		1	1												
	:1·3·1·3:		1								1	1	1	1										1	1											
(h)	:1·3·1·4:		1								1								1			1														
(i)	:1·3·1·3·1:		1											1										1		1		1								
(j)	:1·3·1·3·2:		1																									1	1							
(k)	:1·3·1·3·1·1:		1											1										1				1								
(l)	:2·1:																													1	1					
(m)	:2·2:																													1	1			1		
(n)	:2·3:																													1	1		1			
(o)	:2·4:																													1	1	1	1			
(p)	:2·5:																													1	1	1				
(q)	:2·6:																													1		1				
(r)	:2·7:																													1	1				1	
(s)	:2·8:																													1	1					1

1) OS = <empty>
2) INIT(R,S) is defined to be {if length(R) = 0 then true
   else R.first = S.first ∧ INIT(final(R),final(S))}
   ∧ a "good" sequence is defined to be "a sequence of 1's, 2's, and
   3's containing no adjacent identical subsequences"
   ∧ Q represents the sequence of lexicographically
   ordered "good" sequences
3) there exists a good sequence of length 100
4) requires ability to set S to <empty> and
   length(S) to 0
5) INIT(OS,Q) ∧ OS.last = S
6) (INIT(OS,Q) ∧ OS.last = S ∧ length(S) ≠ 100) ⊃
   INIT(OS⌢"next lexicographically larger good sequence than
   the value in S",Q)
7) requires ability to read length(S)
8) INIT(OS⌢"next lexicographically larger good sequence than
   the value in S",Q)
9) INIT(OS⌢S,Q)
10) requires ability to access each element of S
11) requires ability to write each element of S
12) requires ability to concatenate onto S, i.e. S ← S⌢d
13) requires ability to delete from the end of S, i.e. back from S

14) (Assumptions 1), 2), 3) ⊃ true) ∧
(S = <empty> ∧ length(S) = 0 ⊃ Assumptions 1), 2), 3))
15) INIT(OS⁀"next good sequence after current sequence in S",Q)
∧ INIT(OS,Q) ∧ OS.last = S ⊃
S' equals S prior to executing :1.3: ∧ S is a good sequence
∧ ∀x[S' illt x illt S ⊃ x is not a good sequence]
∧ S is a good sequence ∧ S ≠ S' ⊃
INIT(OS⁀S,Q)
16) (INIT(OS⁀S,Q) ⊃ true) ∧
(OS = OS'⁀S where OS' equals OS prior to :1.4:
⊃ (INIT(OS,Q) ∧ OS.last = S
∧ OS.last = S))
17) S' = S ∧ S is a good sequence
18) initial(S) is a good sequence
19)
20) requires read access to the boolean variable GOOD
21) requires write access to the boolean variable GOOD
22) ∃x[S" illt x ∧ length(x) ≤ length(S") ∧
x is a good sequence] ∧ [(initial(S) is good) ∧ S.last = 0
⊃ S is not good]
23) S" = S
24) ∃x[S" illt x length(x) ≤ length(S")]
25) initial(S) is a good sequence ∧
0 ≤ S.last ≤ 3
26) (Assumptions 2, 13,23,25, 27) ⊃
(Assumptions 2, 13, 23, 27)
27) ability to read S.last
28) ability to write S.last
29) d is an array which contains the digits of S,
one digit per element ∧ the variable length
indexes the last element of S ∧ the sequence is
empty when length = 0
30) ability to read the variable length
31) ability to write the variable length
32) ability to write d[length]
33) ability to read d[length]
34) ability to read every element of d
35) ability to write every element of d

An examination of this table leads to the following decomposition. It shows better entropy loading values than for the table where the implementation information is distributed

((b) ((*) ((d) ((a) ((e) ((h) ((f) (k , j))

.987) .711) .744) .831) .377) .497) .234

*: ((l) ((r) ((s) ((m) ((q) ((n) (o , p))

1.1.) 1.02) .920) .942) .974) .881

This example indicates that complicated interpretations of the contents of variables can lead to high entropy loadings if that information is distributed. The structure can be improved if additional objects are introduced which provide the effects of these complicated assumptions.

HEAPSORT

The sorting algorithm HEAPSORT has been described[WIL], explained[KN2], and verified(via the text of TREESORT3[LO]). However, these descriptions and proofs remain difficult to follow, not because the algorithm is difficult, but because a reader must understand operators which manipulate a tree structure in terms of an implementation of that tree structure as an array. In this example, TREESORT3 is developed by establishing its correctness in terms of operators which manipulate an arbitrary binary tree. The operators and their definitions are a subset of a module definition due to Parnas[PA2]. The definitions are only meant to be descriptions of the general capabilities of the operators which will be implemented. As the development proceeds, information which simplifies these implementations will be distributed and the results will be examined using object/assumption tables and the measure.

LS(i) Left Son(i)

initial value: defined prior to execution of the algorithm below.
effect: error call if there is no definition of the left son of node i; otherwise the name of the left son of node i.

RS(i) Right Son(i)

initial value: defined prior to the execution of the algorithm below.
effect: error call if there is no definition of the right son
of node i; otherwise the name of the right son of node i.

ELS(i) Exists Left Son(i)

possible values: **true, false**
effect: error call if node i has no direct ancestor.

ERS(i) Exists Right Son(i)

possible values: **true, false**
effect: error call if node i has no direct ancestor.

VAL(i) VALue(i)

initial value: set prior to the execution of the algorithm below.
effect: error call if VAL(i) is undefined.

SVA(i,v) Set Value(i,v)

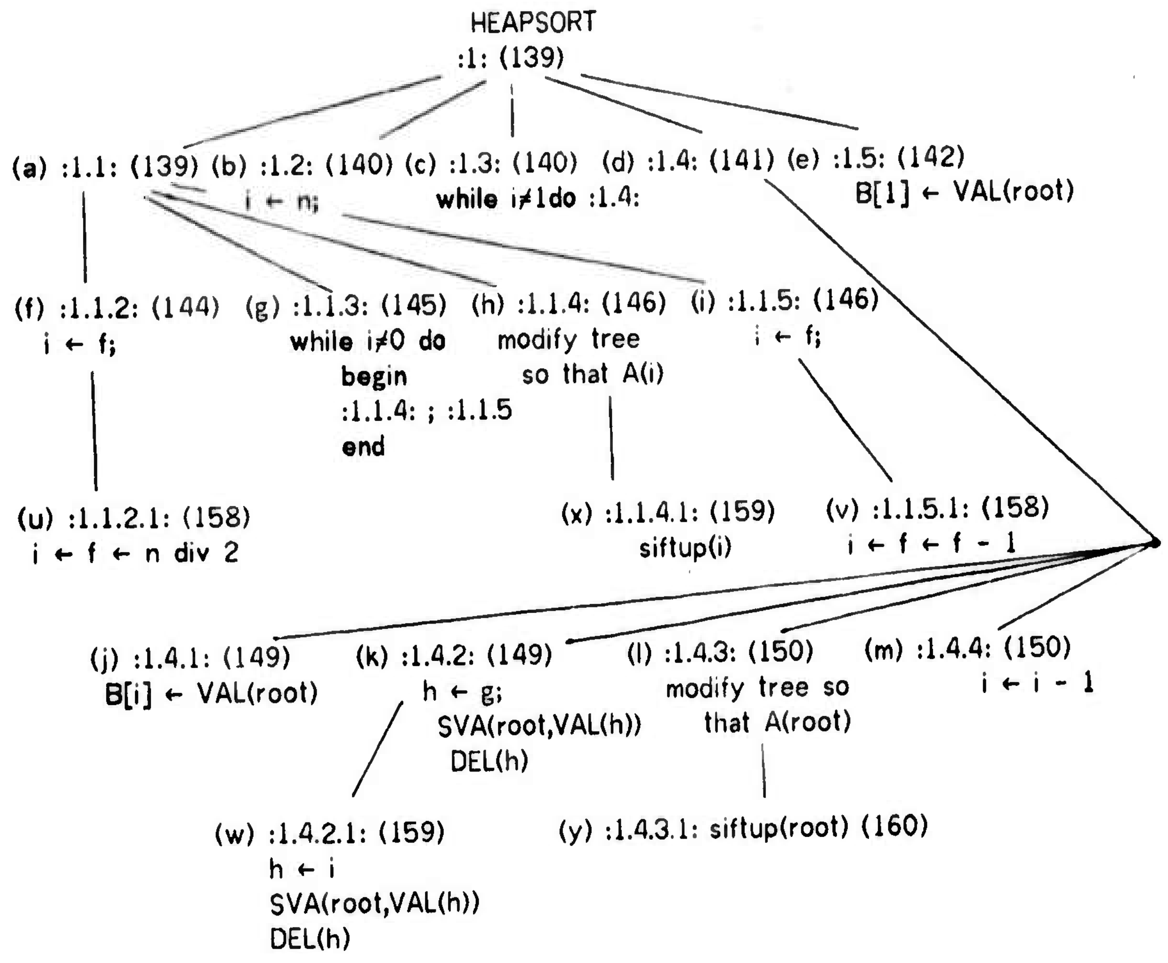
This function has no value.
effect: error call of node i has no direct ancestor;
otherwise VAL(i) = v

DEL(i) DELete(i)

This function has no value.
effect: error call if i has no direct ancestor or
error call if LS(i) or RS(i) are undefined;
otherwise VAL(i) becomes undefined and i is never
again a possible value of RS or LS.

Only after the algorithm is developed, are the implementations of the operators described.

Below is a map describing this development.



(z) :3: (161)
Transform tree so that A(j)
if A(LS(j)) and A(RS(j))

(n) :2.1: (153)
VAL(k), i.e.
VAL ← TREE[k]

(o) :2.2: (154)
SVA(j,k), i.e.
TREE[j] ← x

(p) :2.3: (154)
ERS(j), i.e.
ERS ← if 2*j+1 > NN then
false else true

(q) :2.4: (155)
ELS(j) i.e.
if 2*j > NN then
false else true

(r) :2.5: (155)
RS(j), i.e.
RS ← 2*j + 1

(s) :2.6: (156)
LS(j), i.e.
LS ← 2*j

(t) :2.7: (156)
DEL(j), i.e.
NN ← NN - 1

:1:

assumptions: requires read access to n, the number of nodes in the original tree, requires read access to n, write access required for the elements of array B, requires DEL, requires VAL, requires LS, requires RS, requires ELS, requires ERS

:1: GIVEN A BINARY TREE, HAVING n > 1 NODES AND A SET OF FUNCTIONS: ERS, ELS, VAL, LS, RS, SVA, DEL[PA2], PRODUCE AN ARRAY OF VALUES, B, SUCH THAT THERE IS A ONE TO ONE ONTO MAPPING FROM THE INITIAL VALUES OF THE NODES OF THE TREE TO THE ELEMENTS OF THE ARRAY B AND SUCH THAT THE ELEMENTS OF THE ARRAY ARE ARRANGED IN ASCENDING ORDER.

effects and **post-conditions:** $\forall i[1 \le i < n \supset B[i] \le B[i+1]] \wedge$ there exists a one to one onto mapping from the node values of the initial tree to the elements B[1],...,B[n] of the array

(a) :1.1:

assumptions: A(i) is defined to be $\forall x$[(x is a node of the tree and x = i or x is a descendent of i) $\supset$ ((ELS(x) $\supset$ VAL(x) $\ge$ VAL(LS(x)) $\wedge$ (ERS(x) $\supset$ VAL(x) $\ge$ VAL(RS(x))))] where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that x = C(i), requires ERS, requires ELS, requires RS, requires LS, requires VAL, requires SVA, root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root

:1.1: TRANSFORM THE TREE SUCH THAT A(root) AND THAT THERE EXISTS A ONE TO ONE ONTO MAPPING FROM THE NODE VALUES OF THE INITIAL TREE TO THE CURRENT NODE VALUES OF THE TREE.

effects and **post-conditions:** A(root), there exists a one-one onto mapping of the node values of the initial tree to the current node values

:1.2:

assumptions: requires read access to n, requires write access to the integer variable i, A(root), there exists a one-one onto mapping of the node values of the initial tree to the current node values

```
:1.2:  i ← n;
```

effects: $i = n$

post-conditions: A(root), there exists a one-one onto mapping of the node values of the initial tree to the current node values, $i = n$

(c) :1.3:

assumptions: $\forall j[i < j < n \supset B[j] \leq B[j+1]$, there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], $i < k \leq n$, A(root), $i \geq 1 \wedge$ i equals the number of nodes in the tree $\wedge$ post-conditions for :1.4: $\supset$ assumptions for :1.4: $\wedge$ i is decreased by 1 at each iteration, requires read access to the integer variable i

```
:1.3: while i ≠ 1 do :1.4:
```

effects and **post-conditions:** there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], $i < k \leq n$

(d) :1.4:

assumptions:

A(i) is defined to be ∀x[(x is a node of the tree and x = i or x is a descendent of i) ⊃ ((ELS(x) ⊃ VAL(x) ≥ VAL(LS(x)) ∧ (ERS(x) ⊃ VAL(x) ≥ VAL(RS(x))))] where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that x = C(i), requires ERS, requires ELS, requires RS, requires LS, requires VAL, requires SVA, requires DEL, write access required for the elements of array B, requires read access to the integer variable i, requires write access to the integer variable i, root, names the node such that every node which is not root is a descendent of root ∧ requires read access to root, (ERS(i) ⊃ A(RS(i))) ∧ (ELS(i) ⊃ A(LS(i))), ∀j[i < j < n ⊃ B[j] ≤ B[j+1], there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], i < k ≤ n, A(root)

:1.4: TRANSFORM THE TREE AND B SUCH THAT ∀j[i < j < n ⊃ B[j] ≤ B[j+1]], SUCH THAT THERE EXISTS A ONE TO ONE MAPPING FROM THE INITIAL CONTENTS OF THE TREE TO THE CURRENT CONTENTS OF THE TREE AND THE ELEMENTS B[k], i < k ≤ n, AND THAT A(root).

effects and post-conditions:

∀j[i < j < n ⊃ B[j] ≤ B[j+1], there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], i < k ≤ n, A(root)

(e) :1.5:

assumptions: requires VAL, root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root, write access required for the elements of array B, there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], $i < k \le n$, $\forall j[1 < j < n \supset B[j] \le B[j+1]]$

:1.5: B[1] ← VAL(root);

effects: B[1] = VAL(root)

post-conditions: $\forall i[1 \le i < n \supset B[i] \le B[i+1]]$ $\wedge$ there exists a one to one onto mapping from the node values of the initial tree to the elements B[1],...,B[n] of the array

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	:1:		✓	✓	✓	✓	✓	✓	✓	✓							✓															
(a)	:1.1:	✓	✓	✓	✓	✓	✓	✓								✓																
(b)	:1.2:														✓		✓	✓								✓						
(c)	:1.3:													✓									✓	✓		✓					✓	
(d)	:1.4:	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓				✓			✓	✓		✓						
(e)	:1.5:						✓			✓						✓								✓								✓

1) A(i) is defined to be
$\forall x[(x$ is a node of the tree and $x = i$ or
x is a descendent of $i) \supset$
$((ELS(x) \supset VAL(x) \geq VAL(LS(x)) \wedge$
$(ERS(x) \supset VAL(x) \geq VAL(RS(x))))]$
where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that $x = C(i)$
2) requires ERS
3) requires ELS
4) requires RS
5) requires LS
6) requires VAL
7) requires SVA
8) requires DEL
9) write access required for the elements of array B
13) requires read access to the integer variable i
14) requires write access to the integer variable i
15) root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root
16) requires read access to n
17) there exists a one-one onto mapping of the node values of the initial tree to the current node values
19) $(ERS(i) \supset A(RS(i))) \wedge (ELS(i) \supset A(LS(i)))$
22) $\forall j[i < j < n \supset B[j] \leq B[j+1]$
23) there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and $B[k]$, $i < k \leq n$
25) A(root)
30) $i \geq 1 \wedge i$ equals the number of nodes in the tree $\wedge$ post-conditions for :1.4: $\supset$ assumptions for :1.4: $\wedge$ i is decreased by 1 at each iteration
31) $\forall j[1 < j < n \supset B[j] \leq B[j+1]]$
there exists a one-one onto mapping of the initial tree to the current contents of the tree and $B[k]$, $i \leq k \leq n$

The best decomposition is

((a) ((b) (c , d , e)) 1.33) 1.05

and any further decomposition of c, d, e leads to an entropy loading of 1.61. Hence, for this decomposition, saturation has occurred for (c , d , e). At this stage, the transformation of the tree so that A(root) interacts least with the other objects. Next, :1.1: is elaborated.

(f) :1.1.2:

assumptions: ability to set f such that it has produced no values $\wedge$ the tree is finite $\wedge$ the tree contains at least one node, f is defined to be the value i such that (i has not been produced by a call of f since f was last initialized) otherwise the value of f is 0 $\wedge$ (the nodes of the tree are named by integers which are not equal to 0), requires write access to the integer variable i

:1.1.2: INITIALIZE f; i ← f;

effects and
post-conditions: i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) $\wedge$ i has not held the current value $\wedge$ if i has named all the nodes then i = 0

(g) :1.1.3:

assumptions: requires read access to the integer variable i, root, names the node such that every node which is not root is a descendent of root ∧ requires read access to root, f is defined to be the value i such that (i has not been produced by a call of f since f was last initialized) otherwise the value of f is 0 ∧ (the nodes of the tree are named by integers which are not equal to 0) (if ERS(i) then A(RS(i)) **else true**) ∧ (if ELS(i) **then** A(LS(i)) **else true**) , i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) ∧ i has not held the current value ∧ if i has named all the nodes then i = 0

```
:1.1.3: while i ≠ 0 do

begin

:1.1.4: ; :1.1.5:

end
```

effects: i = 0, i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) ∧ i has not held the current value ∧ if i has named all the nodes then i = 0

post-conditions: there exists a one-one onto mapping of the node values of the initial tree to the current node values, A(root)

(h) :1.1.4:

assumptions:

(ERS(i) ⊃ A(RS(i))) ∧ (ELS(i) ⊃ A(LS(i))), requires read access to the integer variable i, A(i) is defined to be ∀x[(x is a node of the tree and x = i or x is a descendent of i) ⊃ ((ELS(x) ⊃ VAL(x) ≥ VAL(LS(x)) ∧ (ERS(x) ⊃ VAL(x) ≥ VAL(RS(x))))] where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that x = C(i), requires ERS, requires ELS, requires RS, requires LS, requires VAL, requires SVA

:1.1.4: MODIFY THE TREE SUCH THAT A(i) AND THAT THE NODE VALUES ARE PERMUTED.

effects and
post-conditions:

A(i), there exists a one-one onto mapping of the node values of the initial tree to the current node values

(i) :1.1.5:

assumptions:

f is defined to be the value i such that (i has not been produced by a call of f since f was last initialized) otherwise the value of f is 0 ∧ (the nodes of the tree are named by integers which are not equal to 0) (if ERS(i) then A(RS(i)) **else** **true**) ∧ (if ELS(i) **then** A(LS(i)) **else** **true**) , requires write access to the integer variable i

:1.1.5: i ← f;

effects and
post-conditions:

i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) ∧ i has not held the current value ∧ if i has named all the nodes then i = 0

The object assumption table is now

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
:1:		1	1	1	1	1	1	1	1							1																	
(a) :1·1:	1	1	1	1	1	1	1								1																		
(b) :1·2:														1		1	1								1								
(c) :1·3:													1									1	1		1					1			
(d) :1·4:	1	1	1	1	1	1	1	1	1				1	1	1				1			1	1		1								
(e) :1·5:						1			1						1								1								1		
(f) :1·1·2:										1				1							1												
(g) :1·1·3:										1	1		1		1																		1
(h) :1·1·4:	1	1	1	1	1	1	1				1		1																				
(i) :1·1·5:										1	1			1																			

1) A(i) is defined to be
$\forall x[($x is a node of the tree and $x = i$ or
x is a descendent of $i) \supset$
$((ELS(x) \supset VAL(x) \geq VAL(LS(x)) \wedge$
$(ERS(x) \supset VAL(x) \geq VAL(RS(x))))]$
where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that $x = C(i)$

2) requires ERS

3) requires ELS

4) requires RS

5) requires LS

6) requires VAL

7) requires SVA

8) requires DEL

9) write access required for the elements of array B

10) f is defined to be the value i such that
(i has not been produced by a call of f since f was last initialized) otherwise the value of f is 0 $\wedge$
(the nodes of the tree are named by integers which are not equal to 0)

11) $(ERS(i) \supset A(RS(i))) \wedge (ELS(i) \supset A(LS(i)))$

13) requires read access to the integer variable i

14) requires write access to the integer variable i

15) root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root

16) requires read access to n, the number of nodes in the original tree

17) there exists a one-one onto mapping of the node values of the initial tree to the current node values

19) $(ERS(root) \supset A(RS(root))) \wedge (ELS(root) \supset A(LS(root)))$

21) ability to set f such that it has produced no values $\wedge$
the tree is finite $\wedge$
the tree contains at least one node

22) $\forall j[i < j < n \supset B[j] \leq B[j+1]$
23) there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and $B[k]$, $i < k \leq n$
25) A(root)
30) $i \geq 1 \wedge$ i equals the number of nodes in the tree $\wedge$ post-conditions for :1.4: $\supset$ assumptions for :1.4: $\wedge$ i is decreased by 1 at each iteration
31) $\forall j[1 < j < n \supset B[j] \leq B[j+1]]$ there exists a one-one onto mapping of the initial tree to the current contents of the tree and $B[k]$, $i \leq k \leq n$
33) i names a node such that for this execution of the while construct, for all previous values held by i, A(i) $\wedge$ i has not held the current value $\wedge$ if i has named all the nodes then i = 0

RLB and RUB are

RLB: ((a) ((b) (c , d , e)) 1.07) 0.0

RUB: ((a) ((b) (c , d , e)) 1.07) .90

But the acutal elaboration has entropy loadings

((f , g , h , i) ((b) (c , d , e)) 1.49) 1.91

One reason for this marked increase in entropy loading values is that information about the variable i is shared between parts where such sharing did not occur at the last stage. A better decomposition is

((f) ((b) ((e) ((c) ((i) ((d) (g , h))

1.91) 1.73) 1.73) 1.67) 1.49) 1.49

Unfortunately, this decomposition, though better, seems arbitrary. Here, (f) [i ← f] interacts least, ut (g) and (h) that help transform the tree so that A(root), interacts most. In this instance, the decomposition suggested by the last stage is not a good one at this stage.

Next, :1.4: is elaborated.

(j) :1.4.1:

assumptions: write access required for the elements of array B, requires VAL, requires read access to the integer variable i, root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root

:1.4.1: B[i] ← VAL(root);

effects: B[i] = VAL(root)

effects and **post-conditions:** there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], $i < k \leq n$

(k) :1.4.2:

assumptions: requires read access to h, there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], $i < k \leq n$, requires write access to h, root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to root, requires VAL, requires SVA, requires DEL, there exists a function g which names a node, i, such that NOT(ERS(i) $\vee$ ELS(i))

:1.4.2: h ← g; SVA(root, VAL(h)); DEL(h);

effects and **post-conditions:** there exists a one-one onto mapping from the initial nodes of the tree to the current nodes of the tree and B[k], $i \leq k \leq n$

(l) :1.4.3:

assumptions:

$A(i)$ is defined to be $\forall x[(x$ is a node of the tree and $x = i$ or x is a descendent of $i) \supset ((ELS(x) \supset VAL(x) \geq VAL(LS(x)) \wedge (ERS(x) \supset VAL(x) \geq VAL(RS(x))))]$ where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that $x = C(i)$, requires ERS, requires ELS, requires RS, requires LS, requires VAL, requires SVA, $(ERS(i) \supset A(RS(i))) \wedge (ELS(i) \supset A(LS(i)))$

:1.4.3: MODIFY THE TREE SUCH THAT A(root) AND THAT THE NODE VALUES ARE PERMUTED

effects and **post-conditions:**

A(root), $\forall j[i \leq j < n \supset B[j] \leq B[j+1]] \wedge$ there exists a one-one onto mapping of the initial tree to the current contents of the tree and $B[k]$, $i \leq k \leq n$

(m) :1.4.4:

assumptions:

requires read access to the integer variable i, requires write access to the integer variable i, A(root), $\forall j[i \leq j < n \supset B[j] \leq B[j+1]] \wedge$ there exists a one-one onto mapping of the initial tree to the current contents of the tree and $B[k]$, $i \leq k \leq n$

:1.4.4: $i \leftarrow i - 1$;

effects and **post-conditions:**

$\forall j[i < j < n \supset B[j] \leq B[j+1]]$, there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and $B[k]$, $i < k \leq n$, A(root)

Below is the object/assumption table for this elaboration

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(j)	:1.4.1:						1			1				1		1																	
(k)	:1.4.2:						1	1	1							1								1			1	1	1				
(l)	:1.4.3:	1	1	1	1	1	1	1								1				1													
(m)	:1.4.4:													1	1											1							1

1) A(i) is defined to be
$\forall x[(x$ is a node of the tree and $x = i$ or
x is a descendent of $i) \supset$
$((ELS(x) \supset VAL(x) \geq VAL(LS(x)) \wedge$
$(ERS(x) \supset VAL(x) \geq VAL(RS(x))))]$
where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that $x = C(i)$
2) requires ERS
3) requires ELS
4) requires RS
5) requires LS
6) requires VAL
7) requires SVA
8) requires DEL
9) write access required for the elements of array B
13) requires read access to the integer variable i
14) requires write access to the integer variable i
15) root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to the variable root
19) $(ERS(root) \supset A(RS(root))) \wedge (ELS(root) \supset A(LS(root)))$
23) there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and $B[k]$, $i < k \leq n$
25) A(root)
26) there exists a function g which names a node, i, such that NOT(ERS(i) v ELS(i))
27) requires read access to h
28) requires write access to h
32) $\forall j[i \leq j < n \supset B[j] \leq B[j+1] \wedge$
there exists a one-one onto mapping of the initial tree to the current contents of the tree and $B[k]$, $i \leq k \leq n$

RLB and RUB for this elaboration are

RLB: ((f) ((b) ((e) ((c) ((i) ((d) (g , h));

0.0) 1.19) 1.29) 1.16) 1.03) 1.16

RUB: ((f) ((b) ((e) ((c) ((i) ((d) (g , h))

1.77) 1.54) 1.64) 1.59) 1.47) 1.37

The actucal entropy loadings are

((f) ((b) ((e) ((c) ((i) ((j , k , l , m)

(g , h)) 2.15) 1.54) 1.67) 1.89) 1.29) 1.29

A better decomposition is

((f) ((b) ((e) ((i) ((c) ((m) ((k , l) ((j) (g , h))

1.85) 1.77) 1.67) 1.67) 1.54) 1.54) 1.29) 1.29

The algorithm to this stage of development represents a solution to the problem as it was originally posed. Next, an implementation decision is made which has the potential for simplifying the construction of the functions f and g as well as the functions which operate on the tree. This decision represents the original n nodes in the array elements TREE[1],..., TREE[n], where the names of the nodes are their array indices and for $1 < k \leq n$, LS(k div 2) = k if k is even and RS(k div 2) = k if k is odd and VAL(k) = TREE[k]. This representation has several important properties:

(a) ERS(k) ⊃ ELS(k)

(b) If TREE contains $m \geq 1$ elements (nodes) in TREE[1],..., TREE[m] then ERS(k) is defined to be $2*k + 1 \leq m$ and ELS(k) is defined to be $2*k \leq m$ for positive integer k.

(c) if TREE contains $m \geq 1$ elements (nodes) in TREE[1],..., TREE[m] and m is a variable which indicates the number of elements in the tree, DEL(m) is accomplished by the assignment $m \leftarrow m - 1$.

(d) (ELS(k) ⊃ (LS(k) = 2*k)) ∧ (ERS(k) ⊃ (RS(k) = 2*k + 1)).

These properties also simplify the construction of f and g. Specifically, f is called only after a previous call of f, whose value is f', has been used to transform the tree such that A(f'). Hence,

```
:1.1.2: INITIALIZE f; i ← f;
```

can be written as

```
i ← f ← n div 2;
```

and

```
:1.1.5: i ← f;
```

can be written as

```
i ← f ← f - 1;
```

since A(k) is vacuously true if k is a terminal node of the tree.

Similarly, the function g can be defined to equal the value of i as defined in :1.2: and decremented in :1.4.4:

Lastly, the definition of root is 1 which indicates that TREE[1] is the root node of the tree.

These properties together provide definitions for the tree operations as follows.

(n) :2.1:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], assumes read access to TREE, parameter for VAL is always legal

```
:2.1: COMPUTE VAL(k), i.e. VAL ← TREE[k];
```

effects and post-conditions: VAL equals the value of node k in the tree

(o) :2.2:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], assumes write access to TREE, j names an integer in the range 1 to n and it is meaningful to assign x to an element of TREE

:2.2: COMPUTE SVA(j,x), i.e. TREE[j] ← x;

effects and
post-conditions: the value of x has been assigned to node j

(p) :2.3:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], parameter for ERS is a positive integer, requires read access to the variable NN, NN indicates the number of nodes currently in TREE, such that TREE[i], 1 ≤ i ≤ NN is a node of the tree

:2.3: COMPUTE ERS(j) i.e. ERS ← if 2*j + 1 > NN **then false else true;**

effects and
post-conditions: ERS = the value "there exists a right son of j"

(q) :2.4:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], parameter for ELS is a positive integer , requires read access to the variable NN, NN indicates the number of nodes currently in TREE, such that TREE[i], $1 \le i \le$ NN is a node of the tree

:2.4: COMPUTE ELS(j) i.e. ELS ← **if** 2*j > NN **then false else true;**

effects and
post-conditions: ELS equals the value of "there exists a left son of j"

(r) :2.5:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], parameter for RS is legal for the current state of the TREE

:2.5: COMPUTE RS(j), i.e. RS ← 2*j + 1

effects and
post-conditions: RS equals the index of the right son of j

(s) :2.6:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], parameter for LS is meaningful

:2.6: COMPUTE LS(j) i.e. LS ← 2*j

effects and
post-conditions: LS equals the index of the left son of j

(t) :2.7:

assumptions: TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], parameter for DEL always names the current value in NN, requires read access to the variable NN, requires write access to the variable NN, NN indicates the number of nodes currently in TREE, such that TREE[i], 1 ≤ i ≤ NN is a node of the tree

:2.7: COMPUTE DEL(j) i.e. NN ← NN -1;

effects and
post-conditions: node j has been deleted from the tree

The object/assumption table for this elaboration is:

	34	35	36	37	38	39	40	41	42	43	44	45	46
(n) :2.1:	1	1		1									
(o) :2.2:	1		1		1								
(p) :2.3:	1					1					1		1
(q) :2.4:	1						1				1		1
(r) :2.5:	1							1					
(s) :2.6:	1								1				
(t) :2.7:	1									1	1	1	1

34) TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j]
35) assumes read access to TREE
36) assumes write access to TREE
37) parameter for VAL is always legal
38) j names an integer in the range 1 to n and it is meaningful to assign x to an element of TREE
39) parameter for ERS is a positive integer
40) parameter for ELS is a positive integer
41) parameter for RS is legal for the current state of the TREE
42) parameter for LS is legal
43) parameter for DEL always names the current value in NN
44) requires read access to the variable NN
45) requires write access to the variable NN
46) NN indicates the number of nodes currently in TREE, such that TREE[i], $1 \le i \le$ NN is a node of the tree

A good decomposition of this development is

((*) ((f) ((b) ((l) ((i) ((c) ((m) ((k , e) ((j)

(g , h)) 1.42) 1.39) 1.39) 1.28) 1.20) 1.16) .96) .96) .67

where

*: ((n) ((o) ((r) ((s) (t , p , q) .70) .72) .75) .83

The addition of these new objects improves the entropy loadings for the former decomposition.

The objects which elaborate the definitions of f and g are:

(u) :1.1.2.1:

assumptions: f is called only after the immediately preceding call of f, whose value is f', has been used to transform the tree such that A(f') ∧ the sequence of values n div 2, ..., 0 is a sequence of node value which satisfy all the assumptions of :1.1.4:, read access required for f, TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], requires write access to the integer variable i, requires read access to n, the number of nodes in the original tree,

:1.1.2.1: i ← f ← n div 2;

effects and **post-conditions:** i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) ∧ i has not held the current value ∧ if i has named all the nodes then i = 0, (ERS(i) ⊃ A(RS(i))) ∧ (ELS(i) ⊃ A(LS(i)))

(v) :1.1.5.1:

assumptions: f is called only after the immediately preceding call of f, whose value is f', has been used to transform the tree such that A(f') ∧ the sequence of values n div 2, ..., 0 is a sequence of node value which satisfy all the assumptions of :1.1.4:, read access required for f, write access required for f, TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], requires read access to n, the number of nodes in the original tree

:1.1.5.1: i ← f ← f - 1;

effects and **post-conditions:** i names a node such that for this execution of the **while** construct, for all previous values held by i, A(i) ∧ i has not held the current value ∧ if i has named all the nodes then i = 0

(w) :1.4.2.1:

assumptions: requires read access to h, requires write access to h, there exists a one-one mapping from the initial contents of the tree to the current contents of the tree and B[k], i < k ≤ n, root, names the node such that every node which is not root is a descendent of root ∧ requires read access to the variable root, requires VAL, requires SVA, requires DEL, TREE is a one-dimensional array containing n elements, i.e. TREE[1],..., TREE[n], and contains the representation of the tree in the form RS(j) = TREE[2*j + 1] and LS(j) = TREE[2*j], the number of elements in the tree equals the value of i. Hence node i has no descendants, requires read access to the integer variable i

:1.4.2.1: h ← i; SVA(root,VAL(h)); DEL(h);

effects and
post-conditions: there exists a one-one onto mapping from the initial nodes of the tree to the current nodes of the tree and B[k], i ≤ k ≤ n

Lastly, most of the tree operations can be localized to one object by making the following elaborations:

(x) :1.1.4.1:

assumptions: the assumptions of :1.1.4: ⊃ (there exists a procedure, siftup(j), which assumes that (ELS(j) ⊃ A(LS(j))) ∧ (ERS(j) ⊃ A(RS(j))) and results in A(j), (ERS(i) ⊃ A(RS(i))) ∧ (ELS(i) ⊃ A(LS(i))), requires read access to the integer variable i

:1.1.4.1: siftup(i);

effects and
post-conditions: A(i), there exists a one-one onto mapping of the node values of the initial tree to the current node values

(y) :1.4.3.1:

assumptions: the assumptions of :1.4.3: $\supset$ (there exists a procedure, siftup(j), which assumes that (ELS(j) $\supset$ A(LS(j))) $\wedge$ (ERS(j) $\supset$ A(RS(j))) and results in A(j)) , root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to the variable root, (ERS(root) $\supset$ A(RS(root))) $\wedge$ (ELS(root) $\supset$ A(LS(root)))

:1.4.3.1: siftup(root);

effects and **post-conditions:** A(root), $\forall j[i \le j < n \supset B[j] \le B[j+1]$ $\wedge$ there exists a one-one onto mapping of the initial tree to the current contents of the tree and B[k], $i \le k \le n$

(z) :3:

assumptions: (ERS(i) ⊃ A(RS(i))) ∧ (ELS(i) ⊃ A(LS(i))), ERS(i) ⊃ ELS(i), assumes read/write access to NOLOOP, j, t, copy, requires ERS, requires ELS, A(i) is defined to be ∀x[(x is a node of the tree and x = i or x is a descendent of i) ⊃ ((ELS(x) ⊃ VAL(x) ≥ VAL(LS(x)) ∧ (ERS(x) ⊃ VAL(x) ≥ VAL(RS(x))))] where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that x = C(i), requires RS, requires LS, requires VAL, requires SVA, (ELS(j) ⊃ A(LS(j))) ∧ (ERS(j) ⊃ A(RS(j)))

```
:3: siftup(j):
copy ← VAL(j);
repeat
    begin
    NOLOOP ← true;
    if ELS(j) then
        begin
        if ERS(j) then
            begin
            if VAL( RS(j) ) > VAL( LS(j) ) then
                t ← RS(j)
            else
                t ← LS(j)
            end;
         if VAL(t) > copy then
            begin
            SVA(j, VAL(t));
            j ← t;
            NOLOOP ← false;
            end
         end
      end
until NOLOOP;
SVA(j, copy);
```

effects and
post-conditions:
A(the value of j on entry to siftup)

The object assumption table for these additional objects is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
(u) :1·1·2·1:															I		I												
(v) :1·1·5·1:																	I												
(w) :1·1·4·1:												I		I															
(x) :1·4·2·1:							I	I	I					I		I								I				I	I
(y) :1·4·3·1:																I				I									
(z) :3:		I	I	I	I	I	I	I				I																	

	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55		
(u) :1·1·2·1:			I															I	I					
(v) :1·1·5·1:			I															I	I	I				
(w) :1·1·4·1:																					I			
(x) :1·4·2·1:			I																					I
(y) :1·4·3·1:																						I		
(z) :3:																I	I						I	

1) A(i) is defined to be
$\forall x[(x$ is a node of the tree and $x = i$ or
x is a descendent of $i) \supset$
$((ELS(x) \supset VAL(x) \geq VAL(LS(x)) \wedge$
$(ERS(x) \supset VAL(x) \geq VAL(RS(x))))]$
where "x is a descendent of i" means "there exists a composition of the functions LS and RS, say C, such that $x = C(i)$
2) requires ERS
3) requires ELS
4) requires RS
5) requires LS
6) requires VAL
7) requires SVA
8) requires DEL
9) write access required for the elements of array B
10) f is defined to be the value i such that (i has not been produced by a call of f since f was last initialized) otherwise the value of f is 0 $\wedge$ (the nodes of the tree are named by integers which are not equal to 0)
11) $(ERS(i) \supset A(RS(i))) \wedge (ELS(i) \supset A(LS(i)))$
12) there exists a one-one onto mapping of the node values prior to the transformation to the node values after
13) requires read access to the integer variable i
14) requires write access to the integer variable i
15) root, names the node such that every node which is not root is a descendent of root $\wedge$ requires read access to the variable root
16) requires read access to n, the number of nodes in the

original tree

17) there exists a one-one onto mapping of the
node values of the initial tree to the current node values

18) $i = n$

19) $(ERS(root) \supset A(RS(root))) \wedge (ELS(root) \supset A(LS(root)))$

20) $A(i)$

21) ability to set f such that it has produced no values $\wedge$
the tree is finite $\wedge$
the tree contains at least one node

22) $\forall j[i < j < n \supset B[j] \leq B[j+1]$

23) there exists a one-one mapping from the initial contents
of the tree to the current contents of the tree and
$B[k]$, $i < k \leq n$

24) $(ERS(root) \supset A(RS(root))) \wedge (ELS(root) \supset A(LS(root)))$

25) $A(root)$

26) there exists a function g which names a node, i, such
that $NOT(ERS(i) \vee ELS(i))$

27) requires read access to h

28) requires write access to h

29) there exists a one-one onto mapping from the
initial nodes of the tree to
the current nodes of the tree and $B[k]$, $i \leq k \leq n$

30) $i \geq 1 \wedge i$ equals the number of nodes in the tree $\wedge$
post-conditions for :1.4: $\supset$ assumptions for :1.4: $\wedge$
i is decreased by 1 at each iteration

31) $\forall j[1 < j < n \supset B[j] \leq B[j+1]]$

32) $\forall j[i \leq j < n \supset B[j] \leq B[j+1] \wedge$
there exists a one-one onto mapping of the initial
tree to the current contents of the tree and
$B[k]$, $i \leq k \leq n$

33) i names a node such that
for this execution of the while construct,
for all previous values held by i, $A(i) \wedge$
i has not held the current value $\wedge$ if i has named
all the nodes then $i = 0$

34) TREE is a one-dimensional array containing n elements, i.e.
TREE[1],..., TREE[n], and contains the representation of the
tree in the form $RS(j) = TREE[2*j + 1]$ and
$LS(j) = TREE[2*j]$

35) assumes read access to TREE

36) assumes write access to TREE

37) parameter for VAL is always legal

38) j names an integer in the range 1 to n and
it is meaningful to assign x to an element of TREE

39) parameter for ERS is a positive integer

40) parameter for ELS is a positive integer

41) parameter for RS is legal for the current state of the TREE

42) parameter for LS is legal

43) parameter for DEL always names the current value in NN

44) requires read access to the variable NN

45) requires write access to the variable NN
46) NN indicates the number of nodes currently in TREE, such that TREE[i], $1 \le i \le$ NN is a node of the tree
47) ERS(i) $\supset$ ELS(i)
48) assumes read/write access to NOLOOP, j, t, copy
49) f is called only after the immediately preceding call of f, whose value is f', has been used to transform the tree such that A(f') $\wedge$ the sequence of values n div 2, ..., 0 is a sequence of node value which satisfy all the assumptions of :1.1.4:
50) read access required for f
51) write access required for f
52) the assumptions of :1.1.4: $\supset$ (there exists a procedure, siftup(j), which assumes that (ELS(j) $\supset$ A(LS(j))) $\wedge$ (ERS(j) $\supset$ A(RS(j)))
and results in A(j)
53) the assumptions of :1.4.3: $\supset$ (there exists a procedure, siftup(j), which assumes that (ELS(j) $\supset$ A(LS(j))) $\wedge$ (ERS(j) $\supset$ A(RS(j)))
and results in A(j))
54) (ELS(j) $\supset$ A(LS(j))) $\wedge$ (ERS(j) $\supset$ A(RS(j)))
55) the number of elements in the tree equals the value of i. Hence node i has no descendants

A good decomposition for this elaboration is

((y) ((b) ((*) ((z) ((w) ((u , v) ((m ((e) ((c) ((g)

((j , x)) 1.44) 1.38) 1.30) 1.10)

1.19) 1.08) 1.04) 1.01) 1.00) .71

where (*) consists of objects n through t.

This decomposition localizes the tree operations to the objects (*) from the previous decomposition and the uses of the function f to (u) amd (v). Entropy loading figures are higher when information about the implementation of the tree is distributed.

THE PROBLEM OF THE EIGHT QUEENS AND A TELEGRAM PROBLEM: A DISCUSSION

The developments for the GCD Computation, the Sequences Problem, and Heapsort demonstrated applications of the techniques described in Chapters II and III. Similar developments have been constructed for the Eight Queens Problem[W] and a Telegram Problem[HE]. A complete presentation of these developments contributes little to demonstrating the techniques which have already been presented. Instead, the results of these developments are described.

THE PROBLEM OF THE EIGHT QUEENS

The discussion which follows represents an analysis of the development due to Wirth[W], using the measure at each stage. Early stages possess good structure but much information is shared in the final solution. This problem can be stated as:

> Find an arrangement of eight chess queens on an 8 x 8 chess board such that no queen is attacked by any other (i.e. such that each row, column, and diagonal contains at most one queen).

The first stage in Wirth's solution is

```
variable board, pointer, safe;
considerfirstcolumn;
repeat
    begin
    trycolumn;
    if safe then
        begin
        setqueen;
        considernextcolumn
        end
    else
        regress
    end
until lastcoldone or regressoutoffirstcol;
```

This stage is accompanied by the following informal descriptions:

> **considerfirstcol.** The problem essentially consists of inspecting the safety of squares. A pointer variable designates the currently inspected square. The column in which this square lies is called the currently inspected column. This procedure initializes the pointer to denote the first column.
>
> **trycolumn.** Starting at the current square of inspection in the considered column, move down the column either until a safe square is found, in which case the Boolean variable safe is set to **true** or until the last square is reached and is also unsafe, in which case the variable safe is set to **false**.
>
> **setqueen.** A queen is positioned onto the last inspected square.
>
> **considernextcolumn.** Advance to the next column and initialize its pointer of inspection.
>
> **regress.** Regress to a column where it is possible to move the positioned queen further down, and remove the queens positioned in the columns over which regression takes place. (Note that we may regress over at most two columns. Why?)

These informal descriptions do not provide adequate information about the requiresments of each procedure. For example, **considerfirstcolumn** can be interpreted as only requiring that the column pointer be initialized, when in fact the program requires that both the pointer desginating the current square of inspection be initialized as well. This requirement might be suggested by the description of **considernextcolumn** bu not necessarily from the description of **considerfirstcolumn** alone. Thus, the collective descriptions provide the necessary information for implementing all the procedures, but each individual description does not provide enough information for implementing that procedure.

Next, **trycolumn** and **regress** are elaborated. (To this stage, Wirth has made no mention of the requirements for **lastcoldone** or **regressoutoffirstcol**.)

```
procedure trycolumn;
    repeat
        begin
        advancepointer;
        testsquare
        end
    until safe or lastsquare;

procedure regress;
    begin
    reconsiderpriorcolumn;
    if NOT(regressoutoffirstcol) then
        begin
        removequeen;
        if lastsquare then
            begin
            reconsiderpriorcolumn;
            if NOT(regressoutoffirstcol) then
                removequeen
            end
        end
    end;
```

In order for these elaborations to be correct, certain unstated assumptions must be satisfied. Two of these are

(1) Since the first operation in **trycolumn** increments the pointer of inspection, its initial value (set by **considerfirstcolumn** or **considernextcolumn** must have the value that no squares are ignored.

(2) **reconsiderpriorcolumn** must have the effect of extablishing the context of the immediately preceding column.

To this stage, Wirth has carefully represented the solution so that **trycolumn** and **regress** interact little with the main program.

Next, Wirth makes the design decision that the variable j will be the column pointer and the array x[1:8] will be the square pointers.

Thus x[j] is the square pointer for the j-th column, $1 \leq j \leq 8$. Below are the elaborations of these objects along with the assumptions which they make.

:10:

assumptions: j is the column pointer, requires write access to j, assumes the name of the first column is 1, the array x is an array of pointers such that x[j] indicates a square name in column j, requires write access to the array x, the accessed value of x must be set to zero since **trycolumn** will immediately increment it by 1, assumes the name of the first row is 1

```
:10: procedure considerfirstcolumn;
    begin
    j ← 1;
    x[1] ← 0
    end
```

effects and **post-conditions:** j = 1 and the requirements for **trycolumn** are satisfied.

:11:

assumptions: j is the column pointer, requires read access to j, requires write access to j, the array x is an array of pointers such that x[j] indicates a square name in column j, requires write access to the array x, the accessed value of x must be set to zero since **trycolumn** will immediately increment it by 1, assumes the name of the first row is 1, assumes **considernextcolumn** will be invoked only when there is a column named j + 1

```
:11: procedure considernextcolumn;
    begin
    j ← j + 1;
    x[j] ← 0;
    end;
```

effects and **post-conditions:** j is incremented by 1 and the requirements for **trycolumn** are satisfied.

:12:

assumptions: j is the column pointer, requires read access to j, requires write access to j, assumes reconsiderpriorcolumn will be invoked only when there is a column named j - 1

```
:12: procedure reconsiderpriorcolumn;
j ← j - 1;
```

effects and **post-conditions:** the column pointer has been set to the immediately preceding column

:13:

assumptions: j is the column pointer, requires read access to j, the name of the next square in a column equals the current square name plus 1, assumes advancepointer will only be called if there exists a "next" square in the current column, the array x is an array of pointers such that x[j] indicates a square name in column j, requires read access to the array x, requires write access to the array x

```
:13: procedure advancepointer;
x[j] ← x[j] + 1;
```

effects and **post-conditions:** x[j] has been incremented by 1 to name the next square in column j

:14:

assumptions: assumes the number of rows is 8, the array x is an array of pointers such that x[j] indicates a square name in column j, requires read access to the array x, j is the column pointer, requires read access to j

```
:14: Boolean procedure lastsquare;
lastsquare ← x[j] = 8;
```

effects and **post-conditions:** lastsquare = **true**, if the last square in the current column is named by the square pointer; **false** otherwise.

:15:

assumptions: j is the column pointer, requires read access to j, assumes the number of column is 8

:15: **Boolean procedure** lastcoldone;
lastcoldone ← j > 8;

effects and
post-conditions: lastcoldone = **true** if the column pointer names a column which is greater than the name of the last column.

:16:

assumptions: j is the column pointer, requires read access to j, assumes the name of the first column is 1

:16: **Boolean procedure** regressoutoffirstcol;
regressoutoffirstcol ← j > 1;

effects and
post-conditions: regressoutoffirstcol = **true**, if the column pointer names a column which is less than the name of the first column

Next, Wirth observes that by introducing the Boolean arrays a, b, and c with the meanings

a[k] = **true**: no queen is positioned in row k,
b[k] = **true**: no queen is positioned in /-diagonal k, and
c[k] = **true**: no queen is positioned in \-diagonal k.

testsquare, **setqueen**, and **removesquare** can easily be implemented if the index ranges for a, b, and c are chosen carefully. The observation that 1, ... , 8 names the eight rows suffices for the range of k for a. Further, since the sums of the subscripts for squares on a board (board[1:8, 1:8]) in the /-diagonals is unique for each diagonal and identical for each square in a single /-diagonal, an appropriate subscript range for b is 2, ... , 16. Similarly, the difference of the subscripts (first subscript minus second subscript) for \-diagonals is unique for each \-diagonal and identical for each square in a \-diagonal. This suggests that the subscript range for c should be -7, ... , 7.

Consequently, the elaborations:

:17:

assumptions: j is the column pointer, requires read access to j, requires write access to Boolean variable safe, assumes the number of rows is 8, assumes the name of the first row is 1, the array x is an array of pointers such that x[j] indicates a square name in column j, requires read access to the array x, the sum of the indices in a single /-diagonal are identical and lie in the range 2,...,16; the sum of the indices in a \-diagonal are identical and lie in the range -7,...,7; , a[k] = **true**: no queen is positioned in row k, read access required for a, b[k] = **true**: no queen is positioned in /-diagonal k, read access to b required, c[k] = **true**: no queen is positioned in \-diagonal k, read access to c required

```
:17: procedure testsquare;
safe ← a[x[j]] ∧ b[j + x[j]] ∧ c[j - x[j]];
```

effects and **post-conditions:** safe = **true**, if in column j, square x[j] is not attacked by any queen in columns 1, ... , j-1

:18:

assumptions: j is the column pointer, requires read access to j, assumes the number of rows is 8, assumes the name of the first row is 1, the array x is an array of pointers such that x[j] indicates a square name in column j, requires read access to the array x, the sum of the indices in a single /-diagonal are identical and lie in the range 2,...,16; the sum of the indices in a \-diagonal are identical and lie in the range -7,...,7; , a[k] = **true**: no queen is positioned in row k, write access required for a, b[k] = **true**: no queen is positioned in /-diagonal k, write access to b required, c[k] = **true**: no queen is positioned in \-diagonal k, write access to c required

```
:18: procedure setqueen;
a[x[j]] ← b[j + x[j]] ← c[j - x[j]] ← false;
```

effects and
post-conditions: a queen is positioned in column j, square x[j], and its influence in the appropriate row, and the two diagonals is set

:19:

assumptions: j is the column pointer, requires read access to j, the array x is an array of pointers such that x[j] indicates a square name in column j, assumes the number of rows is 8, assumes the name of the first row is 1, requires read access to the array x, the sum of the indices in a single /-diagonal are identical and lie in the range 2,...,16; the sum of the indices in a \-diagonal are identical and lie in the range -7,...,7; , a[k] = **true**: no queen is positioned in row k, write access required for a, b[k] = **true**: no queen is positioned in /-diagonal k, write access to b required, c[k] = **true**: no queen is positioned in \-diagonal k, write access to c required

```
:19: procedure removequeen;
a[x[j]] ← b[j + x[j]] ← c[j - x[j]] ← true;
```

effects and
post-conditions: a queen is removed from column j, as well as the appropriate row and diagonals

Below is the object assumption table for these parts.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
:10:	I		I	I		I	I					I	I													
:11:	I	I	I	I		I	I						I	I												
:12:	I	I	I																							I
:13:	I	I		I	I	I		I	I																	
:14:	I	I		I	I					I																
:15:	I	I									I															
:16:	I	I										I														
:17:	I	I		I	I					I			I		I	I	I	I		I	I		I	I		
:18:	I	I		I	I					I			I			I	I		I	I		I	I		I	
:19:	I	I		I	I					I			I			I	I		I	I		I	I		I	

1) j is the column pointer
2) requires read access to j
3) requires write access to j
4) **the** array x is an array of pointers such that x[j] indicates **a** square name in column j
5) requires read access to the array x
6) requires write access to the array x
7) the accessed value of x must be set to zero since **trycolumn** will immediately increment it by 1
8) the name of the next square in a column equals the current square name plus 1
9) assumes advancepointer will only be called if there exists a "next" square in the current column
10) assumes the number of rows is 8
11) assumes the number of column is 8
12) assumes the name of the first column is 1
13) assumes the name of the first row is 1
14) assumes **considernextcolumn** will be invoked only when there is a column named j + 1
15) requires write access to Boolean variable safe
16) the sum of the indices in a single /-diagonal are identical and lie in the range 2,...,16; the sum of the indices in a \\-diagonal are identical and lie in the range -7,...,7;
17) a[k] = **true**: no queen is positioned in row k
18) read access required for a
19) write access required for a
20) b[k] = **true**: no queen is positioned in /-diagonal k
21) read access to b required
22) write access to b required
23) c[k] = true: no queen is positioned in \\-diagonal k
24) read access to c required
25) write access to c required
26) assumes reconsiderpriorcolumn will be invoked only when there is a column named j - 1

In the absence of the remaining portions of the table, (objects and assumptions for the main program, **trycolumn**, and **regress**) it is still meaningful to compute entropy loadings for the portion displayed. This is justified because objects :10: through :19: do not share specific assumptions with the main program or **trycolumn**, or **regress**. Consequently, the best decomposition to this stage, must involve two large parts - :10: through :19: and the main program, **trycolumn**, **regress** - which are then further decomposed. A good decomposition for the displayed objects is:

((:15:) ((:16:) ((:12:) ((:13: , :14:) ((:10: , :11:) ((:17:)

((:18: , :19:)) 1.83) 1.64) 1.42) .94) .64) .64

Better entropy loading figures can be found by noting that :10: (**considerfirstcol**) and :11: (**considernextcol**) share information not only about j but also about x. By introducing a new object :20: (**initsqofinspect**), and modifying :10: and :11:, a decomposition where :10: and :11: emerge sooner as a subset of a good decomposition can be found.

Below are the relevant objects:

:10:

assumptions: j is the column pointer, requires write access to j, assumes the name of the first column is 1, requires ability to invoke **initsqofinspect** which sets the square of inspection to a value which is the proper initialization for examining the squares in a column named by j

```
:10: procedure considerfirstcolumn;
    begin
    j ← 1;
    initsqofinspect;
    end
```

effects and **post-conditions:** $j = 1 \wedge x[1] = 0$, $x[1]$ can be incremented by 1 to satisfy the requirements for **trycolumn**

:11:

assumptions: j is the column pointer, requires read access to j, requires write access to j, requires ability to invoke **initsqofinspect** which sets the square of inspection to a value which is the proper initialization for examining the squares in a column named by j, assumes **considernextcolumn** will be invoked only when there is a column named j + 1

```
:11: procedure considernextcolumn;
    begin
    j ← j + 1;
    initsqofinspect;
    end;
```

effects and **post-conditions:** j is incremented by 1 and the square pointer for the next column equals 0, to satisfy the requirements for **trycolumn**.

:20:

assumptions: j is the column pointer, requires read access to j, the array x is an array of pointers such that x[j] indicates a square name in column j, requires write access to the array x, the accessed value of x must be set to zero since **trycolumn** will immediately increment it by 1, assumes the name of the first row is 1

```
:20: procedure initsqofinspect;
x[j] ← 0;
```

effects and **post-conditions:** x[j] = 0 and can be incremented by 1 to satisfy the requirements for **trycolumn**.

The object/assumption table is now:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
:10:	1		1									1															1
:11:	1	1	1											1													1
:12:	1	1	1																							1	
:13:	1	1		1	1	1		1	1																		
:14:	1	1		1	1					1																	
:15:	1	1									1																
:16:	1	1										1															
:17:	1	1		1	1					1			1		1	1	1	1		1	1		1	1			
:18:	1	1		1	1					1			1			1	1		1	1		1	1		1		
:19:	1	1		1	1					1			1			1	1		1	1		1	1		1		
:20:	1	1		1		1	1						1														

1) j is the column pointer
2) requires read access to j
3) requires write access to j
4) the array x is an array of pointers such that x[j] indicates a square name in column j
5) requires read access to the array x
6) requires write access to the array x
7) the accessed value of x must be set to zero since **trycolumn** will immediately increment it by 1
8) the name of the next square in a column equals the current square name plus 1
9) assumes advancepointer will only be called if there exists a "next" square in the current column
10) assumes the number of rows is 8
11) assumes the number of column is 8
12) assumes the name of the first column is 1
13) assumes the name of the first row is 1
14) assumes **considernextcolumn** will be invoked only when there is a column named j + 1
15) requires write access to Boolean variable safe
16) the sum of the indices in a single /-diagonal are identical and lie in the range 2,...,16; the sum of the indices in a \-diagonal are identical and lie in the range -7,...,7;
17) a[k] = **true**: no queen is positioned in row k
18) read access required for a
19) write access required for a
20) b[k] = **true**: no queen is positioned in /-diagonal k
21) read access to b required
22) write access to b required
23) c[k] = **true**: no queen is positioned in \-diagonal k
24) read access to c required
25) write access to c required
26) assumes reconsiderpriorcolumn will be invoked only when there is a column named j - 1

27) requires ability to invoke **initsqofinspect** which sets the square of inspection to a value which is the proper initialization for examining the squares in a column named by j

The former decomposition, grouping :20: with :10: and :11: is

((:15:) ((:16:) ((:12:) ((:13: , :14:) ((:10: , :11: , :20:)

((:17:) (:18: , :19:)) 1.77) 1.59) 1.39) .86) .60) .60

A better decomposition is

((:15:) ((:16:) ((:12:) ((:10: , :11:) ((:13: , :14:) ((:20:)

((:17:) (:18: , :19:)) 1.77) 1.59) 1.39) .66) .86) .60) .60

Wirth concludes the development by observing that read accesses to x occur more frequently than write accesses to x (This is apparent from the table, so long as :13: and :21: are executed less frequently than all the objects which read x.) Consequently, it is suggested that since array accesses are usually more costly than access to simple variables, a new variable, i, can be introduced such that x[j] ← i; is always executed before j is incremented and i ← x[j]; is executed after j is decreased. The effect of introducing this change and distributing information about i, is to increase the entropy loadings of the above decomposition.

Lastly, in the final version of the main program, information about i, x, and j is freely distributed. This causes the main program to interact with objects with which it did not interact at earlier stages. As a result, entropy loading figures for the decomposition become larger, and in some cases lead to saturation where it did not occur at earlier stages. Further, changes in the meaning or use of x or j imply that many changes will have to be made throughout the program.

A TELEGRAM PROBLEM

Henderson and Snowdon[HS] hae proided a development and an analysis of a program which was produced using the techniques of structured programming. The program, however, was shown to contain at least one "bug". The authors claim that one stage in the development of more information about the program environment was distributed than was necessary. As a result, the programmer forgot some of this detail at a later stage, thus causing the error. This observation suggests that objects in the development were allowed to interact more than was necessary. It also suggests that by not explicitly obsering assumptions, programmers construct assumptions which may or may not be correct. The discussion below first states the problem and then presents the deelopment due to Henderson and Snowdon [HE] to the point where the error occurred. Shared assumptions are emphasized.

The Problem

A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits, and blanks on some deice and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word "ZZZZ". The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength

> words. The words "ZZZZ" and "STOP" are not chargeable and words of more than twelve characters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

Before proceeding, it should be noted that the description of the problem is not as precise as it should be. Aside from an incomplete description of the specific behavior of commands which invoke input operations as well as operations which select single characters from the buffer, the definition of a "word" is not precise enough. The strings "ZZZZ" and "STOP" are not chargeable words, but from the program which is presented, a telegram consisting of zero or more occurrences of "STOP" followed by "ZZZZ" is considered to be an empty telegram. This interpretation is not consistent with this author's understanding of the statement of the problem. Nevertheless, Henderson and Snowdon develop a solution as follows (The object names have been added in order to clarify the ancestry of objects.)

```
:1.1: INITIALIZE FOR WHOLE PROGRAM;

:1.2: repeat
    begin
    :1.3: INITIALIZE FOR NEW TELEGRAM;
    :1.4: PROCESS TELEGRAM
    end
until EMPTY TELEGRAM;
```

This program requires that at least one telegram be part of the input and that an empty telegram must occur. It seems questionable whether :1.1: and :1.3: should be stated in this stage. As with version III of the GCD computation, such initializations seem more natural if they emerge as a result of satisfying the assumptions of certain objects.

Next, :1.4: PROCESS TELEGRAM, is elaborated as

:1.4.1: COUNT, CHECK, AND PRINT WORDS;

:1.4.2: PRINT WORD COUNT AND CHECK MESSAGE;

Object :1.4.2: assumes that a count of the number of words and a check for overlength words is available. Further, it is assumed that :1.4.1: provides this information. Hence, in the absence of an explicit attempt to hide the mechanisms which provide this information :1.4.1: and :1.4.2: must share several assumptions. All assumptions about telegram syntax and information about what is to be recorded for each telegram is contained in :1.4.1:. No assumptions about the explicit manner of inputting text have yet been made.

The elaboration of :1.4.1: is

```
:1.4.1.1: repeat
    begin
    :1.4.1.2: EXTRACT WORD;
    :1.4.1.3: if WORD IS CHARGEABLE then
        :1.4.1.4: COUNT WORD;
    :1.4.1.5: if WORD IS TOO LONG then
        :1.4.1.6: SET CHECK FLAG;
    :1.4.1.7: PRINT WORD;
    end
 until WORD IS "ZZZZ";
```

:1.4.1.2: requires information about what constitutes a word, i.e. the next sequence of non-blank characters. :1.4.1.3: requires information about which words are chargeable, i.e. words which are not "STOP" or "ZZZZ". :1.4.1.5: requires the information about what constitutes an overlength word. Objects :1.4.1.4: and :1.4.1.6: require variables that reflect the state of the number of words in the current telegram and whether any overlength words have occurred in this telegram. At this point, the values of these variables are observed to require some kind of initialization. Thus, :1.3:, :1.4.1.4: and :1.4.1.6: share assumptions.

Object :1.4.1.2:, EXTRACT WORD, is elaborated as

```
:1.4.1.2.1: SET WORD EMPTY INITIALLY;

:1.4.1.2.2: ADJUST INPUT;

:1.4.1.2.3: repeat
    :1.4.1.2.4: EXTRACT LETTER
until LETTER IS SPACE;
```

This elaboration is really the source of the error which occurs in the final program. Until this stage, all assumptions have been concerned with the properties of telegrams, but :1.4.1.2.2:, ADJUST INPUT, necessarily introduces assumptions about the way input is performed or at least about how the buffer is managed. Similarly, :1.4.1.2.4:, EXTRACT LETTER, makes some of these assumptions. :1.4.1.2.4:. This implies that these objects share assumptions which are not directly related to the task of extracting the next word from the telegram. The descriptions are also not precise. The authors comment that the condition

first letter of input $\neq$ space

must hold prior to the execution of :1.4.1.2.3:. Ths condition is not necessarily suggested as the effect of the phrase, ADJUST INPUT.

A clearer elaboration might be

```
repeat
    EXTRACT LETTER
until LETTER IS NOT A SPACE;

SET WORD TO EMPTY;

repeat
    begin
    CONCATENATE LETTER TO THE RIGHT END OF WORD;
    EXTRACT LETTER
    end
until LETTER IS A SPACE;
```

(Implicit in both elaborations is the assumption that a space always follows a word, even if a letter is the last character of the entire input file. The authors solve this difficulty by concatenating a space to the end of each input record.) This second elaboration localizes all assumptions about handling input to EXTRACT LETTER and its elaborations. As a result, entropy loading figures for decompositions of the program involving this second elaboration are generally lower than for the original program.

CHAPTER V

ON ASPECTS OF USING THE MEASURE: SUMMARY, EVALUATION, CONCLUSIONS

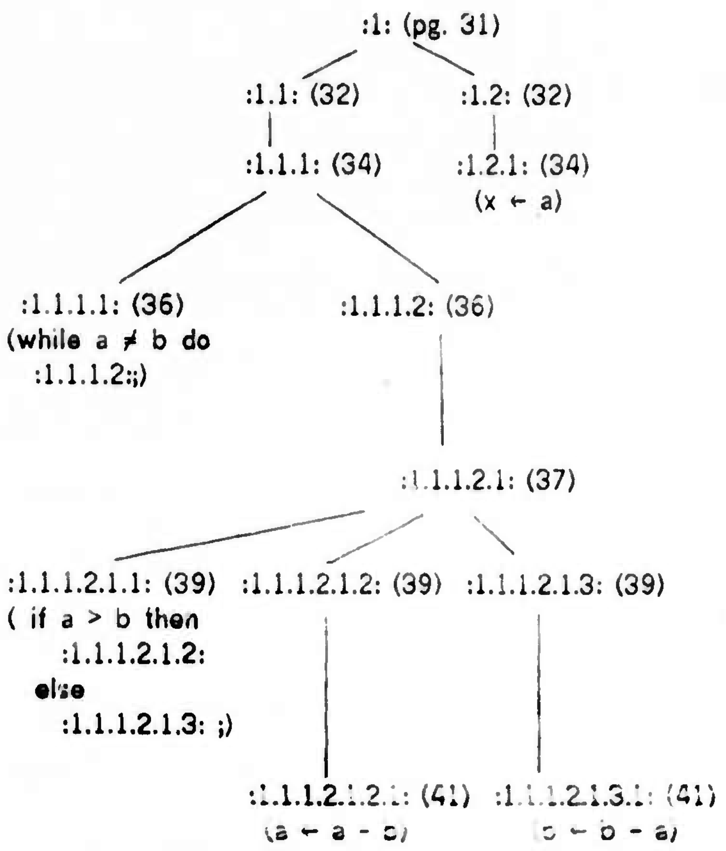
Chapter IV presented examples of how entropy loading calculations could be used as guides to help control program structure. (Appendix I applies these techniques to a larger program.) This chapter first summarizes the major results of each example. Next, the potential advantages of using the methodology and the measure are stated. Using the measure in a practical situation, however, poses certain difficulties. These are listed and form the basis for several suggestions for future research.

REVIEW OF RESULTS DEMONSTRATED BY THE EXAMPLES

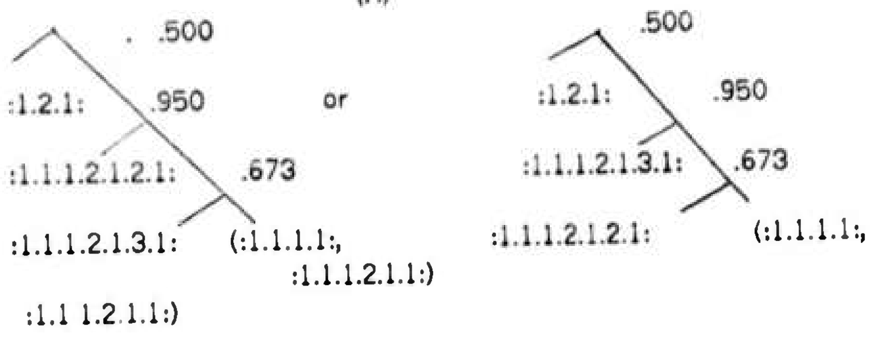
This thesis has investigated the question of whether a particular methodology describes program structure as defined in Chapter II. The methodology under investigation uses a mathematical calculation, called entropy loading, in two ways. First, given a development of a program where the assumptions have been preserved at each stage, entropy loading figures can compare different arrangements of objects in an attempt to discover which groupings of objects interact least. Such decompositions might suggest ways for constructing a set of modules whose combined effects solve the original problem. Second, if a particular decomposition is suggested at early stages in a development, entropy loading figures can be used to observe whether the development at later stages still possesses similar structural properties. If good structure is not preserved, the object/assumption table (in which the assumptions are preserved) might suggest ways of localizing certain assumptions to existing or new objects.

In Chapter IV, three versions of a program that computes the greatest common divisor are analyzed. The map for version I and the trees corresponding to two decompositions of the final program appear below.

A GCD COMPUTATION



(A)



(B)

. .500

:1.2.1: 1.05

(:1.1.1.1:, :1.1.1.2.1.1:) (:1.1.1.2.1.2.1:,
:1.1.1.2 1.3.1)

This analysis corresponds to an attempt to find the best way of decomposing this fixed program. The measure indicated that the best decomposition of the program is decomposition (A). The part which interacts least with the rest of the program is :1.2.1: that assigns the value of a to x. Objects :1.1.1.2.1.2.1: and :1.1.1.2.1.3.1: assign values to a and b and also require more information than :1.2.1: but less information than :1.1.1.1: and :1.1.1.2.1.1:. These last objects determine the flow of control within the program. This result is consistent with the definition of structure and our intuitive ideas. In this example, the assumptions associated with the control mechanisms include information about what is being controlled. No attempt was made to hide that information. However, the objects being controlled were constructed without making assumptions about mechanisms that control them. Decomposition (B) is slightly worse because the statements (a ← a - b, b ← b - a) require read and write access to both a and b. Hence, more information is localized to a single subset than in (A) where write access to a and b are separated.

Version II is similar to version I except that the construction

```
if a > b then a ← a - b else b ← b - a;
```

is replaced by

```
while a > b do a ← a - b;
while b > a do b ← b - a;
```

Here, the best decomposition,

((a) ((g , h) (b) ((c , e)) 1.01) 1.01) .451

indicates that (a) [x ← a] interacts least with the rest of the program and that of the remaining portion, (g , h) (a ← a - b, b ← b - a) interacts least with the control mechanisms b (while a ≠ b do) and (c , e) (while a > b do, while b > a do).

Version III illustrates a development similar to version II, but in addition to computing the greatest common divisor of a and b, their least common multiple is also computed. Here, the decomposition

((a) ((b) ((c) ((k , l) (g , i)) 1.55) 1.75) 1.75) 1.28

indicates that (a) [x ← a; y ← c + d;] interacts least with the rest of the program, but that the control mechanisms for the inner loops as well as the statements which they control interact most. Object b (c ← 0; d ← a) initializes c and d, but really shares little information with the rest of the program. Similarly, object c (while a ≠ b do) controls the inner loops but interacts little with the mechanisms which decrease a and b. Note that the entropy loading figures for this decomposition are larger than the corresponding figures for version II. This occurs because the objects in version III make assumptions that are shared in more complicated ways than the objects in version II.

The development of the sequences problem shows that distributed information about a scheme for representing data can lead to unnecessarily complicated structure. This example produces a list of lexicographically ordered sequences such that each sequence contains only 1's, 2's and 3's, but no adjacent identical subsequences. The list is terminated by the first such sequence containing 100 digits. The development to the stage where information about the representation of a sequence in terms of an array resulted in the decomposition

((b) ((d) ((a) ((e) ((h) ((f) (k , j))

1.21) 1.21) 1.91) 1.38) 1.21) 1.21

Entropy loadings for this decomposition are all greater than or equal to entropy loadings for the same oecomposition where implementation information was not distributed. This suggests that the implementation information be localized to one or several objects. In this case, additional objects were introduced. This resulted in entropy loadings that were smaller and very close to the entropy loadings for the decomposition prior to elaborating the implementation. Thus, the measure provided indications that motivated a rearrangement of the program so that the reasonably good structure of the early stages was preserved in the final program.

The example that develops a sorting program based on the algorithm HEAPSORT attempts to present a program in terms of the model that probably motivated the algorithm: a binary tree representing the data to be sorted. One reason for attempting this exercise was to first describe the algorithm without presenting all the details of its implementation. Then a particular representation for the binary tree - as a linear array - is introduced. As a consequence of this decision, information about the representation can be localized to certain objects without adversely affecting the decompositions suggested at earlier stages. Although applications of the measure eventually suggested a decomposition that localized almost all assumptions about the representation to one object (:3:, the siftup procedure), earlier decompositions had to be discarded. One reason for this occurrence is that the small number of objects make many assumptions. As a result, saturation occurred at several early stages. Further, since none of the assumptions were weighted with "probability of change" figures, some of the decompositions seemed to be counter to the author's view of what a good decomposition should be. For example, the two objects invoking the procedure siftup do not form a single subset in the decomposition that is presented - probably because one call of siftup uses i as its parameter, and i is used throughout the program, where the other call uses root as its parameter, and root is used in few places.

The example that presents the Eight Queens Problem shows several improvements in a program that already possesses fairly good structure.

First, the procedures **considerfirstcolumn** and considernextcolumn not only share information about how the columns of the chess board are arranged and named, but also about certain requirements of another object, **trycolumn**. By creating a new object, **initsqofinspect**, a decomposition that was already rather good was slightly improved. In Wirth's original program, the objects **considernextcolumn** and **considerfirstcolumn** not only share information about the names and ordering of columns of squares on a chess board, but also about how the squares are represented and about some of the assumptions made by **trycolumn**. **initsqofinspect** contains these assumptions about **trycolumn** and thus helps to improve the structure of the program. The final version of Wirth's program distributed information that was not shared at earlier stages, thus making these decompositions worse than they need be. This corresponds to the similar situation in the development of the sequences problem. Saturation was also apparent at early stages.

The discussion of the telegram problem was presented in order to emphasize the importance of precisely stated assumptions. Henderson and Snowdon[HE] have stated that informal English comments are not sufficient to suggest the assumptions which objects make or the affects they are intended to produce. For example, the condition

first letter of input ≠ space

is not necessarily suggested by the phrase "ADJUST INPUT". Such imprecision is mentioned as a potential source for errors in a program. The example also cites a portion of the development where apparently too much detail was introduced. The elaboration of :1.4.1.2:, EXTRACT WORD, introduces an object that caters to the requirements of the input device. This implies that the elaboration of ADJUST INPUT and EXTRACT LETTER will probably share assumptions about the nature of the input device. An alternative is presented that localizes all this information to EXTRACT LETTER. The entropy loading figures, though not displayed, indicate that this new version possesses better structure.

Appendix I applies the measure to a development of a Markov Algorithm interpreter. Although the analysis of the example is lengthy, several results can be stated here:

> (1) The structure imposed at the initial stages, i.e. alphabet and generic input, algorithm input, error handling during input, algorithm execution, and error handling during execution, could be maintained in the final version if additional objects were introduced to provide access to the results of these additional objects. For example, objects which created internal representations of rules were introduced along with accessors to this information. These emerged together in a subset after those subsets that constituted the structure at the initial stages.
>
> (2) Due to the large number of objects and assumptions, several clerical aids had to be used extensively. These aids, and suggestions for extending them, are described in a later section.
>
> (3) As in the development of HEAPSORT, several good decompositions found at intermediate stages had to be discarded. This was necessary because later elaborations resulted in objects that shared much information with objects that existed prior to the elaboration. For example, the rule input portion was elaborated after the portion that handled alphabets and generic input. The earlier decomposition had to be modified. A similar situation occurred when the algorithm execution part was elaborated.

REVIEW OF RESULTS DEMONSTRATED BY THE EXAMPLES

Appendix II discussed a paper on Compiler Structure[M K]. This paper asserted that special care should be taken to describe the languages at the interfaces between vertically fragmented modules in a compiler. However, because of the results from the Markov Algorithm Interpreter, it was concluded that as much information about these languages should be hidden. Instead, only creators and accessors of the needed information should be provided.

ADVANTAGES OF USING THE MEASURE

This thesis has applied a measurement function based on entropy loadings to evaluate decompositions of programs. These applications produced results that usually corresponded to intuitive ideas about structure in programs. Unfortunately, there are practical problems about deciding the relative importance of assumptions as well as problems about determining and manipulating assumptions and tables. These problems impede the effective use of the measure and methodology. Details of the advantages and disadvantages of using the measure are discussed in this and the next section.

In order to apply the measure, the methodology requires that the objects and assumptions be explicitly stated - not only at early stages in the design process, but also in the final program. Ideally each object is accompanied by the assumptions it makes so that the object is understandable without requiring additional context. This feature is typically no present in the more traditional approaches to design and programming. Further, the assumptions made by objects are summarized in object/assumption tables. This summary makes it possible to observe the assumptions that are shared among objects without needing to deduce them from the program text. This is especially helpful whenever entropy loading figures become larger than anticipated. Under these circumstances, other decompositions are suggested more readily than in situations where object/assumption tables are not available.

The measure provides a way of comparing different decompositions of a program at each stage in its development. These comparisons help to substantiate decisions to reject or retain a decomposition. As a result, there are at least quantitative grounds for arguing for or against a decomposition rather than primarily intuitive ones. The sequences example emphasizes this point.

The act of "hiding" information can be explicitly represented by the methodology. Hidden information is preserved in object/assumption tables.

Decompositions resulting from applications of the measure can possess some of the properties advocated by Parnas[PA1,PA2,PA3]. In particular, subsets that share few assumptions can suggest modules similar to those described by Parnas. For example, the analysis of the Markov Algorithm interpreter suggests that objects which create and manipulate Markov rules should appear in a single subset even though they are invoked from portions which interact little. Further, the descriptions of objects required by the methodology can help to suggest specifications similar to those proposed by Parnas. One characteristic of these specifications is that each describes the intended behavior of a function without stating an algorithm that implements it. Many of these behavioral descriptions occur among the assumptions of objects at early stages in the development of a program. These can be used to help generate Parnas-type specifications. As an example, below is a specification for a function which might be used by the first version of the GCD computation

```
function PGCD(a,b) - a and b are integer parameters.

    effects: ERC1 if a ≤ 0
             ERC2 if b ≤ 0
             gcd(a,b) = gcd(a',b') ∧ ((a' ≠ b') ⊃
                                  (a < a' or b < b'))
```

This specification was found by using tests for the assumptions to suggest error calls and describing the effects of objects as part of the effects of the function. An important property of the error calls is that each represents some testable condition. Some assumptions may not represent such conditions. Since predicates relating to the correct behavior of objects appear as assumptions, the objects can be constructed with these assumptions in mind. Specifically, a designer/programmer will be, perhaps, more conscious of the explicit demands his objects must meet. The error that occurred in the Telegram Problem might not have occurred had these assumptions been explicit.

Again because assumptions are presented explicitly, changes in a program that require violations of some assumptions can be made more easily than under circumstances where assumptions are not stated. A programmer need only examine the object/assumption table to determine which assumptions are violated. As a result, the affected objects can be changed. The overhead of deducing the effects of changes from program text and other traditional aids is eliminated.

DIFFICULTIES OF USING THE MEASURE

It should be apparent that entropy loading calculations can be computed easily once the assumptions made by objects have been displayed in an object/assumption table. Finding these assumptions, however, is often a painstaking process. This process is made even more difficult by what seems to be a natural tendency to postpone the task of stating assumptions. As a result, the task becomes more difficult because the assumptions made by earlier stages must be deduced from a context that is different from the context that motivated those stages. Most often this exercise of stating assumptions and using the measure to check various decompositions leads to results that might already have been expected. The interesting cases, of course, are those where this exercise led to unexpected results or actually uncovered an error. The decompositions found during the middle middle stages in the development of HEAPSORT as well as during the middle stages of the Markov Algorithm Interpreter did not possess good structure at later stages. This occurred because objects that were elaborated after those stages shared many assumptions with earlier elaborations of objects. Lastly, instances where several designers are elaborating different objects or where objects are elaborated without a knowledge of other objects often lead to incompatible representations of similar assumptions. Consequently, interactions may not be represented properly in object/assumption tables.

Of the different kinds of assumptions, the most difficult to state are weakest pre-conditions for objects. This difficulty is not surprising in the light of all the practical difficulties associated with program verification. Once found, however, these assumptions provide vital information about the requirements of objects. Assumptions about program environment are more easily recognized, but can frequently be overlooked. For example, a designer working on an elaboration of a single object might make an assumption so frequently that he omits it from the object/assumption table. This could lead to an eventual decomposition where this unstated assumption is violated. Probably the easiest assumptions to state are the mathematical theorems relevant to the problem and the assumptions about data. The theorems are often related to the weakest pre-conditions. The assumptions about data refer to those items that are explicitly stated in many informal descriptions of objects and relate to items that are analogous to what will be manipulated in the language which implements the program.

DIFFICULTIES OF USING THE MEASURE

In addition to the difficulties in stating assumptions, there are difficulties involved in just manipulating tables and selecting decompositions for which to compute entropy loadings. Object/assumption tables can become large even for small numbers of objects.

In chapter III, certain kinds of object/assumption tables were cited for which all decompositions had identical entropy loadings. Hence, the measure was unable to distinguish among them. This was called saturation in object/assumption tables. Saturation occurs most frequently whenever a small number of objects share many assumptions. The developments of HEAPSORT and the Markov Algorithm Interpreter displayed instances of saturation.

In all but the simplest situations, it is difficult to assign "probability of change" figures or "relative importance" to assumptions. Consequently, all the assumptions in the examples were treated as though they were of equal importance.

Without the help of mechanical aids, the process of constructing programs using the methodology and the measure is tedious and time consuming. An experienced programmer might be able to construct programs having good structure in far less time.

AIDS TO APPLYING THE MEASURE AND SUGGESTIONS FOR FUTURE WORK

As aids to help solve some of the difficulties stated in the last section, several programs have been constructed. These programs perform the following tasks:

(1) input and maintain files of assumptions

(2) input and maintain abbreviated descriptions of objects and their assumptions.

(3) produce object/assumption tables given files generated by (1) and (2).

(4) produce listings of objects given files generated by (1) and (2).

(5) compute entropy loading calculations, RLB's, and RUB's given object/assumption tables produced by (3).

These programs have been used to help produce all the examples in Chapter IV ano in the appendices. Each is intended to help solve some clerical or tedious aspect of using the measure. As the examples indicate, a great deal of text might need to be manipulated for even small programs. These programs have been written to execute in an interactive environment. This has proved to be helpful when entropy loading calculations were performed. Values for RLB, RUB and the actual entropy loadings could be compared quickly in this kind of environment.

However, in order to use the measure in more realistic and practical situations, the following topics suggest areas for future research:

(1) Since assumptions about the meaning and interpretation of variables occur so frequently, and since explicitly transcribing them - or their names - to identified objects is tedious and time consuming, mechanical aids should be available that allow a programmer to state these assumptions only once - perhaps as part of some declaration. Then, the program support should automatically associate the appropriate given specific constructs and specific post-conditions.

(2) Assumptions about control and frequency of use of objects should be expressable precisely.

(3) Because the process of making assumptions can require careful and time consuming thought, as many assumptions as possible should be generated mechanically.

(4) Because of the substantial difficulties associated with applying the measure, additional measures of structure should be sought.

CONCLUSIONS

The purpose of this thesis has been to help clarify the notion of structure in programs and to evaluate the behavior of a particular measure of structure. The most difficult aspects of applying the measure relate to the process of making assumptions explicit. Many of these difficulties can be obviated by mechanical aids that can be part of the environment in which the measure is to be used. The information about programs provided by the measure makes possible comparisons of different decompositions of a program. This is demonstrated in Chapter IV and the appendices. If interactions are more extensive than is desirable, the object/assumption table tells exactly which assumptions are shared and can suggest that certain assumptions be localized to new or existing objects. Parnas-type specifications seem to be deducible in a direct way from the assumptions made by objects and their effects.

This thesis has attempted to demonstrate, use, and evaluate a definition and measure of program structure. It represents an attempt to extend the notion of structure from its role as an aesthetic tool to a useful and measurable aid for finding good programs. Despite several significant shortcomings, the measure provides a quantitative valuation of a heretofor vague concept.

BIBLIOGRAPHY

[AL] Alexander, C. **Notes on the Synthesis of Form,** Harvard University Press, 1964

[BA] Baker, F.T. Chief programmer team management of production programming. **IBM Systems Journal** v.11,no.1, January, 1972, pp. 56-73

[COO] Cook, S.A. The complexity of theorem proving procedures. **Proc. of the Third Annual Symposium on Theory of Computing,** ACM, May, 1971, pp. 151-157

[DJ1] Dijkstra, E.W. Notes on structured programming, **Structured Programming,** Academic Press, 1972

[DJ2] Dijkstra, E.W. A constructive approach to the problem of program correctness. **BIT** 8 (1968) pp.174-186

[DJ3] Dijkstra, E. W. EWD316: **A Short Introduction to the Art of Programming,** Eindhoven, The Netherlands, 1971

[DJ4] Dijkstra, E.W. The structure of the T.H.E. multiprogramming system. **CACM** 11 (1968) pp. 341-346

[DJ5] Dijkstra, E.W. On the axiomatic definition of semantics. (unpublished ms.)

[DJ6] Dijkstra, E.W. Go to statement considered harmful. **CACM** 11, 3 (March 1968)

[H] Heymanns, F. A Markov Algorithm Interpreter, (unpublished program)

[HE] Henderson, P. and R. Snowdon, An experiment in structured programming. **BIT** 12 (1972), pp. 38-53

[HO1] Hoare, C.A.R. An axiomatic basis for computer programming. **CACM** 12 (1969) pp. 576-580

[HO2] Hoare, C.A.R. Proof of a program: FIND **CACM** 14 (January 1971) pp.39-45

[HO3] Hoare, C.A.R. Notes on data structuring. **Structured Programming,** Academic Press, 1972

[KA] Karp, R.M. Reducibility among combinatorial problems. **Complexity of Computer Computations**, R.E. Miller and J.W. Thatcher, eds., Plenum Press, 1972

[KI] King, J.C. **A Program Verifier**, Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1969

[KN1] Knuth,D.E. **Fundamental Algorithms**, Addison-Wesley, 1968, pp. 187-189

[KN2] Knuth, D.E. **Sorting and Searching**, Addison-Wesley, 1972, pp. 145-147

[LE] Leavenworth, B. Review of paper by P. Naur ["Programming by action clusters." BIT 9, 3 (1969), 250-258]. **Computing Reviews** 11, 7 (July 1970), Rev. 19,420

[LO] London, R.L. Certification of algorithm 245 Treesort3: proof of algorithms - a new kind of certification. CACM 13 (1970) pp.371-373

[MI1] Mills, H.D. Top down programming in large systems. **Debugging Techniques in Large Systems**, R. Rustin, ed., Prentice-Hall, 1970, pp. 41-55

[MI2] Mills, H.D. **Structured Programming**, IBM: Federal Systems Division, Gaithersburg, 1970

[MK] McKeeman, W.M., Horning, J., and Wortman, D.B. **A Compiler Generator**, Prentice-Hall, 1970

[MK1] McKeeman, W.M. **Compiler Structure**, Technical Report CSRG-23, University of Toronto, January, 1973

[NA2] Naur, P. Programming by action clusters. BIT 9 (1969) pp.250-258

[PA1] Parnas, D.L. **Information Distribution Aspects of Design Methodology**, Proceedings of the IFIP Congress, 1971

[PA2] Parnas, D.L. **A Technique for Software Module Specification with Examples**, CACM 15, (May 1972) pp 330-336

[PA3] Parnas, D.L. **On the Criteria to be Used in Decomposing Systems into Modules**, CACM 15 (December 1972) pp 1053-1058

[PA4] Parnas, D.L. Some Conclusions from an Experiment in Software Engineering Techniques, FJCC, 1972

[SN] Snowdon, R.A. PEARL: An Interactive System for the Preparation and Validation of Structured Programs. Computing Laboratory, University of Newcastle upon Tyne, 1971

[vE1] van Emden, M.H. An Analysis of Complexity, Ph. D. Thesis, Mathematisch Centrum, Amsterdam, 1971

[vE2] van Emden, M.H. The heirarchical decomposition of complexity. Machine Intelligence 5, Dale and Michie, eds., Edinburgh University Press, 1970, pp. 361-380

[WA] Watanabe, S. Information theoretical analysis of multivariate correlation. IBM Journal of Research and Development, 1960, pp.66-82

[WIL] Williams, J.W.J. Heapsort. CACM 7 (1964) pp.347-348

[WL] Williams, W. and Lambert, J.M. Multivariate methods in plant ecology I. The Journal of Ecology, 47, pp. 83-101

[W] Wirth, N. Program development by stepwise refinement. CACM 14, (April 1971), pp. 221-227

[W1] Wirth, N. PL/360, A programming language for the 360 computers. J. ACM 15, 1 (January 1968) pp. 37-74

[WRH] Wulf, W.A., Russell, D.B. and Habermann, A.N. BLISS: A language for systems programming. CACM 14, 12 (December 1971) pp. 780-790

APPENDIX I: A MARKOV ALGORITHM PROCESSOR

This appendix contains a development of a Markov algorithm processor, based upon an initial and informal description. The development shows how assumptions can be preserved in a somewhat larger example than any which appeared in Chapter II. The guidelines stated in Chapter III are applied at various stages of the development. The result includes a complete program which inputs and interprets labelled and unlabelled Markov algorithms and several modifications to the basic program. These modifications represent several reasonable changes which can easily be made in the basic program, but which would be difficult or tedious to make in other representations of the same program.

The following document serves as the basis for the development that follows.

MARKOV ALGORITHM INTERPRETER

The interpreter is to be designed so that it executes both labelled and unlabelled Markov algorithms. If the algorithm is not terminated by the dot convention, then execution should halt when the final rule of the algorithm is not applicable. A "blank" character is retained as the first character of the register so that all append rules are applicable.

II. Data File

A. Parameter Card - This must be the first card of the file of input cards. If the card is blank, the register's contents will be printed only at the termination of the algorithm. Its contents will be printed after the execution of each applicable rule if a non-blank character appears in column 1.

B. Header Card - This card indicates the title of the algorithm, the alphabets, and the generic variables (if any) for each alphabet. The syntax of the heading is given in V. There are no format restrictions.

C. Algorithm - A single replacement rule is punched on each card. Its format is as follows:

Cols.	1-3	Right-justified integer label. Since this label has significance only in the control of a labelled Markov algorithm, it is optional in the case of an unlabelled one.
Col 4		A colon
Cols.	5-80	<Markov Rule> <Successor>;

D. Data - A single data card containing an initial character string must be supplied for each execution of the algorithm. A semicolon indicates the termination of the character string. As many data cards as desired may be included.

E. Algorithm Terminator - A card with a comma (,) punched in the first column indicates: (1) the termination of the data for an algorithm, and (2) an additional algorithm will follow.

F. End-of-file - A card with commas punched in both columns one and two terminates the data for the last algorithm in the file.

III. Sample Data File

The following unlabeled Markov algorithm reverses the order of the characters initially placed in the register. The first character of each line, which is assumed to represent a single card, is assumed to be the character punched in the first column of the card. (Note: The labels are optional in this case.)

```
REVERSE(A,B,C,D,E,F);G,H;(+,-);;

006:++:-;

001:-G:G-;

002:-+:-;

003:-:.;

004:+HG:G+H;

005::+;
```

IV. Restrictions

A. The maximum number of alphabets is 10.

B. The maximum number of characters in any alphabet is 30.

C. The maximum number of generic variables, over all alphabets, is 10.

D. The algorithm must have labels in the range 1 to 100.

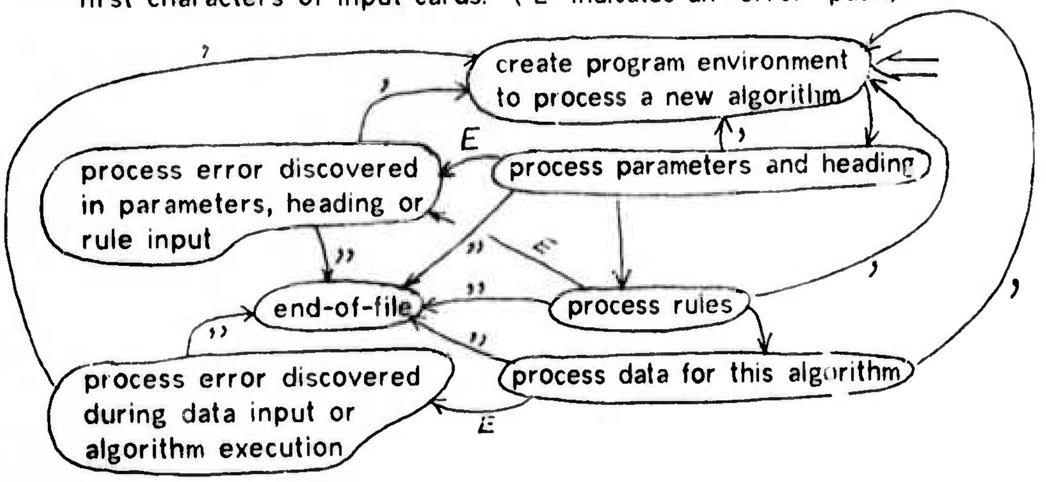
E. The register will hold a maximum of 500 characters.

V. Markov Algorithm Syntax

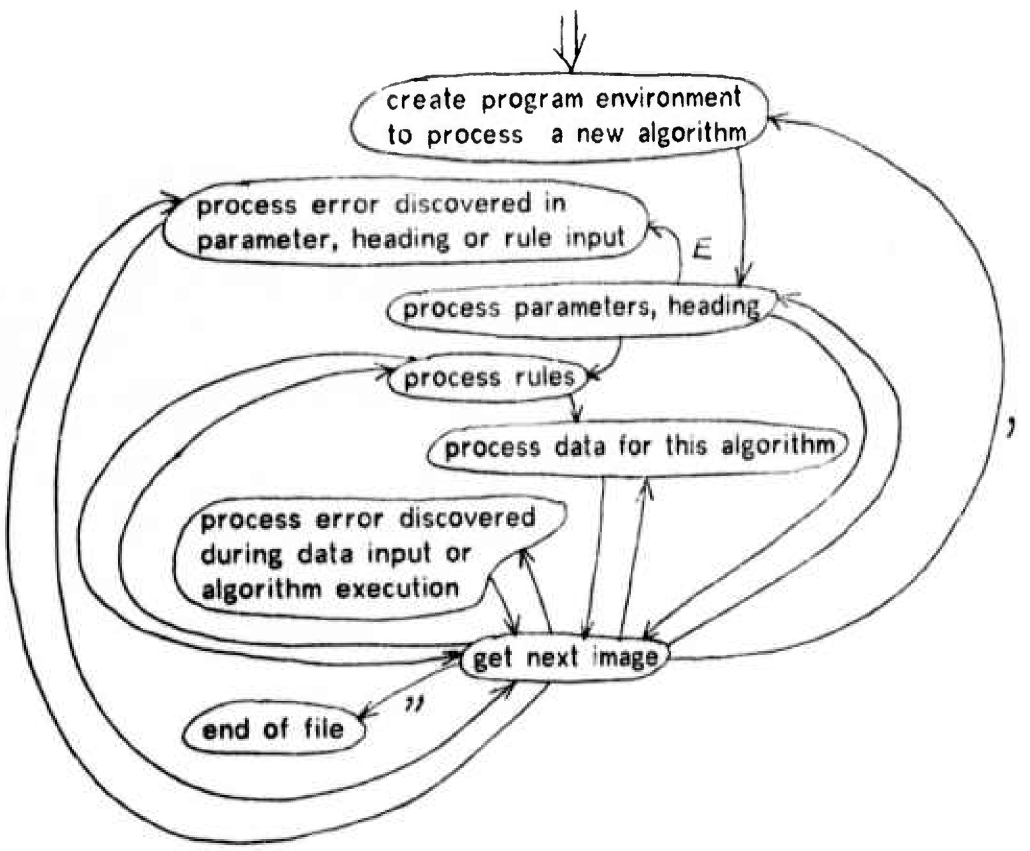
```
<Algorithm> ::= <Heading> <Body>

<Heading> ::= <Title> <Declaration>

<Title> ::= <Character string>

<Declaration> ::= (<Alphabet>); <Generic Declaration>; |
                  <Declaration>
                  (<Alphabet>); <Generic Declaration>;

<Alphabet> ::= <Character> | <Alphabet>,<Character>

<Generic Declaration> ::= <empty> | <Generic list>

<Generic list> ::= <Character> |
                   <Generic list>,<Character>

<Body> ::= <Rule>; | <Body> <Rule>;

<Rule> ::= <Label> : <Markov Rule> <Successor>

<Label> ::= <Digit> | <Label> <Digit>

<Markov Rule> ::= <Side> : <Side>

<Side> ::= <empty> | <Character String>

<Character String> ::= <Character> |
                       <Character String> <Character>

<Successor> ::= <empty> | . | , <Label>

<Character> ::= <all characters which can be punched
                into a card except ",", ":", ".">
```

The following development is based on an unpublished program due to F. Heymanns[H]. In its original form, many assumptions were shared throughout the program, thus making it difficult to understand and change. The fundamental ideas, however, are sound. What follows is an attempt to use these ideas but to preserve the initial structure in the final program. The attempt emphasizes the use of these ideas. Some readers, however, may be able to construct other programs having better structure.

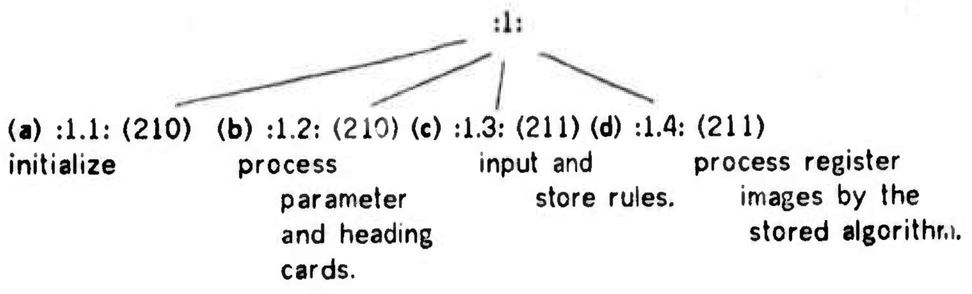
The measure is used at various stages to suggest possible decompositions. One good decomposition seems to possess many of the properties suggested by Parnas as being important in a system. That decomposition bears only a superficial resemblance to an initial decomposition which is described in the next paragraph. Below is an initial decompostion of the Markov algorithm processor represented as a transition diagram. The arcs indicate paths for error conditions as well as transitions corresponding to the occurrence of commas as the first characters of input cards. ("E" indicates an "error" path.)


create program environment to process a new algorithm
process parameters and heading
process error discovered in parameters, heading or rule input
end-of-file
process rules
process data for this algorithm
process error discovered during data input or algorithm execution
E
,
,,

Note that each state requires a knowledge that cards containing a comma in column one and commas in columns one and two force the processor to be initialized to accept a new algorithm or to terminate. Implicit in the diagram is the ability of each state to input a card image. Many of the interactions caused by this shared information can be eliminated by removing the ability to input a card image to a new state, and to include in this state all information regarding the meaning of cards containing commas in the first two columns. This choice is a direct application of the "information hiding " principle which is justified by the value of the measure. Below is the new state diagram.


create program environment to process a new algorithm
process error discovered in parameter, heading or rule input
E
process parameters, heading
process rules
process data for this algorithm
,
process error discovered during data input or algorithm execution
get next image
end of file
"

A map for the first stages in this development appears below.


:1:
(a) :1.1: (210) initialize
(b) :1.2: (210) process parameter and heading cards.
(c) :1.3: (211) input and store rules.
(d) :1.4: (211) process register images by the stored algorithm.

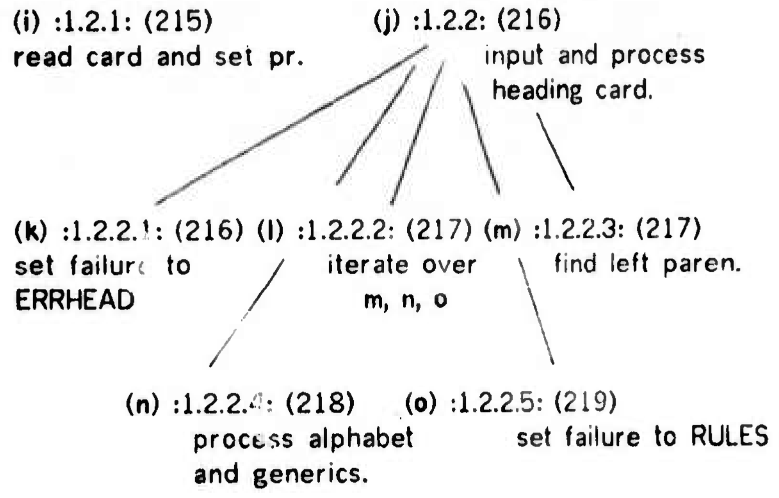
:1.5: (211) process end-of-file condition

(f) :2: (212) read and store a card image

(g) :3: (213) process error occurring in algorithm input part

(h) :4: (213) process error occurring in algorithm execution part

(i) :1.2.1: (215) read card and set pr.


(j) :1.2.2: (216) input and process heading card.
(k) :1.2.2.1: (216) set failure to ERRHEAD
(l) :1.2.2.2: (217) iterate over m, n, o
(m) :1.2.2.3: (217) find left paren.
(n) :1.2.2.4: (218) process alphabet and generics.
(o) :1.2.2.5: (219) set failure to RULES

(a) :1.1:

assumptions: :1.1: PROVIDE ALL NECESSARY INITIALIZATIONS;

effects and **post-conditions:** all necessary initializations have been made

(b) :1.2:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR

:1.2: INPUT AND PROCESS THE PARAMETER CARD;
INPUT AND PROCESS THE HEADING CARD;

effects and **post-conditions:** parameter card and heading card have been correctly processed

(c) :1.3:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR

:1.3: INPUT AND PROCESS THE RULES FOR THIS ALGORITHM;

effects and **post-conditions:** all rules for this algorithm have been inputted

(d) :1.4:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR

:1.4: PROCESS THE DATA IMAGES WITH RESPECT TO THE STORED ALGORITHM;

effects and **post-conditions:** all data images for this algorithm have been processed

(e) :1.5:

assumptions: an end-of-file condition has occurred

:1.5: PROCESS END-OF-FILE CONDITION AND TERMINATE;

effects and **post-conditions:** the Markov algorithm processor has been terminated

(f) :2:

assumptions: requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE, requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm, CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0, write access required for CP, C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE, requires read access to C, requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted, "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm is to be processed

:2: READ A CARD AND STORE THE 80 CHARACTERS IN SUCCESSIVE LOCATIONS OF THE ARRAY C, I.E. C[1],...,C[80] I.E. GETIMAGE.

```
if C[1] = COMMA then
  begin
  if C[2] = COMMA then invoke end-of-file state
 (:1.5:)
    else invoke algorithm initialization state
  end
else CP ← 0
```

effects and **post-conditions:** a new card image has been read and the appropriate transition made.

(g) :3:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, requires ability to print the string argument which is passed as the parameter to ERROR, ability to perform printing operations

```
:3: AN ERROR WAS DISCOVERED BY THE ALGORITHM INPUT
PART;
INDICATE THE ERROR;
READ AND PRINT ALL THE REMAINING IMAGES FOR THIS
ALGORITHM;
while true do
  begin
  READCARD;
  PRINTCARD
  end;
```

effects and **post-conditions:** string parameter has been printed along with all remaining card images for the algorithm being processed

(h) :4:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, requires ability to invoke :1.4: which processes the remaining data images for this algorithm, ability to perform printing operations

```
:4: AN ERROR WAS DISCOVERED BY THE ALGORITHM
EXECUTION PART;
INDICATE THE ERROR;
READ A NEW DATA CARD ;
INVOKE THE ALGORITHM EXECUTION PART;
```

effects and **post-conditions:** a new data image has been read and the execution part has been processed

The object/assumption table for these objects is:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(a)	:1.1:															
(b)	:1.2:	1	1	1												
(c)	:1.3:	1	1	1												
(d)	:1.4:		1	1												
(e)	:1.5:				1											
(f)	:2:					1	1	1	1	1	1	1	1			
(g)	:3:		1											1	1	
(h)	:4:		1	1											1	1

1) requires the ability to invoke :3:, ERROR, with a string message
2) requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available
3) NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR
4) an end-of-file condition has occurred
5) requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE
6) requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm
7) CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0
8) write access required for CP
9) C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE
10) requires read access to C
11) requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted
12) "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm is to be processed
13) requires ability to print the string argument which is passed as the parameter to ERROR
14) ability to perform printing operations
15) requires ability to invoke :1.4: which processes the remaining data images for this algorithm

A good decomposition for these objects is

((a) ((e) ((f) ((g , h) ((d) (b , c)) .97) 1.08) .14) .20) 0

Here, objects (g) and (h) process errors, objects (b) and (c) input the algorithm and object (d) interprets it. Not surprisingly, the lack of assumptions made by the initialization part leaves the entropy loading figure at 0. Next, :1.2: is elaborated

(i) :1.2.1:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, requires write access to PR, pr = **true** means "print the register **after** each successful application of a rule; otherwise d not print the register after each successful application of a rule

```
:1.2.1: READ A NEW CARD IMAGE;
SET THE FUNCTION pr TO INDICATE WHETHER OR NOT THE
REGISTER CONTENTS SHOULD BE PRINTED, I.E.

NEXTCARD;
pr ← if NEXTCHAR = "1" then true else false;
```

effects and **post-conditions:** pr = **true** if the register is to be printed after each successful rule application; **false** otherwise

(j) :1.2.2:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, MAXA equals the maximum number of alphabets permitted for an algorithm, requires read access to MAXA, ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR

:1.2.2: INPUT AND PROCESS THE HEADING CARD IMAGE;

effects and **post-conditions:** The heading card image has been correctly processed

(k) :1.2.2.1:

assumptions: ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR, requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available,

:1.2.2.1: SET THE FAILURE ROUTINE FOR NEXTCHAR TO BE ERRHEAD; NEXTCARD;

effects and **post-conditions:** ERRHEAD has been set as the failure routine for NEXTCHAR and a new card image has been read

(l) :1.2.2.2:

assumptions: a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, either no alphabets have been processed of all alphabets processed have been correct

```
:1.2.2.2:

repeat
  begin
  :1.2.2.3: ;
  :1.2.2.4: ;
  :1.2.2.5
  end
until false;
```

effects and **post-conditions:** The heading card has been correctly processed and is error free.

(m) :1.2.2.3:

assumptions: a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", assumes write access to CHAR, assumes read access to CHAR, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image

```
:1.2.2.3: SCAN FOR A LEFT PARENTHESIS, I.E.
repeat
    CHAR ← NEXTCHAR
until CHAR = OPEN;
```

effects and **post-conditions:** CHAR is an open parenthesis.

(n) :1.2.2.4:

assumptions: ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR, assumes write access to CHAR, assumes read access to CHAR, requires the ability to invoke :3:, ERROR, with a string message, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, MAXA equals the maximum number of alphabets permitted for an algorithm, requires read access to MAXA, NA equals the number of alphabets which have been processed thus far for the current algorithm, requires read accesss to NA, requires write access to NA, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", ERRHEAD assumes compete control when invoked and handles error messages and further processing, requires ability to invoke STORGEN which stores the content of CHAR, if legal, otherwise invokes the apporpriate error, requires ability to invoke STORALPH, which stores the alphabet character if all requirements are met,otherwise STORALPH invokes appropriate error routines, requires ability to invoke ALPHFIN which cmpletes any needed processing after an entire alphabet has been stored

:1.2.2.4: PROCESS AN ALPHABET AND ITS GENERICS;

effects and **post-conditions:** A single alphabet and its generics has been correctly processed

(o) :1.2.2.5:

assumptions: ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR, requires ability to invoke RULES which names the rule input part, but since RULES is a label in the main program a **go** to statement can be used

:1.2.2.5: SETFAIL(RULES);

effects and **post-conditions:** failure routine for NEXTCHAR has been set to the next stage of processing

Below is map for the elaboration of :1.2.2.4: and several objects used by the elaboration.

(p) :1.2.2.4.1: (220)	(q) :1.2.2.4.2: (221)	(r) :1.2.2.4.3: (222)
process alphabet part.	process generic part.	terminate processing for an alphabet.

(s) :11: (225) get next character

(t) :12: (226) store a generic for the current alphabet

(u) :13: (227) store a character into the current alphabet

(v) :14: (228) terminate processing for current alphabet

(w) :15: (228) test whether space exhausted.

(x) :16: (229) test whether character is a legal alphabetic or generic

(y) :17: (229) test whether generic has already been used

(z) :18: (230) test whether character has already occurred in this alphabet.

(p) :1.2.2.4.1:

assumptions: ERRHEAD assumes compete control when invoked and handles error messages and further processing, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, NA equals the number of alphabets which have been processed thus far for the current algorithm, requires read accesss to NA, MAXA equals the maximum number of alphabets permitted for an algorithm, requires read access to MAXA, requires write access to NA, assumes write access to CHAR, assumes read access to CHAR, requires the ability to invoke :3:, ERROR, with a string message, ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", requires read/write access to NOIT, which controls a loop that process alphabets, requires ability to invoke STORALPH, which stores the alphabet character if all requirements are met,otherwise STORALPH invokes appropriate error routines

:1.2.2.4.1: PROCESS ALPHABET PART -

```
if NA + 1 >  MAXA then ERROR("TOO MANY ALPHABETS");
SETFAIL(ERRHEADNAME);
NOIT ← false;

repeat
  begin
  CHAR ← NEXTCHAR;
  STORALPH;
  CHAR ← NEXTCHAR;
  if CHAR ≠ COMMA then
    begin
    if CHAR ≠ CLOSE or NEXTCHAR ≠ SEMI then
      ERRHEAD
    else
      NOIT ← true
    end
  end
until NOIT;

NA ← NA + 1;
```

effects and

post-conditions: a single alphabet has been successfully processed and stored, NA equals the number of alphabets which have been processed thus far for the current algorithm

(q) :1.2.2.4.2:

assumptions: assumes write access to CHAR, assumes read access to CHAR, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, requires read/write access to NOIT1, which controls a loop that processes generics, ERRHEAD assumes compete control when invoked and handles error messages and further processing, requires ability to invoke STORGEN which stores the content of CHAR, if legal, otherwise invokes the apporpriate error

:1.2.2.4.2: PROCESS GENERIC VARIABLES FOR THIS ALPHABET -

```
CHAR ← NEXTCHAR;
NOIT1 ← false;
if CHAR ≠ SEMI then
  begin
  repeat
    begin
    STORGEN;
    CHAR ← NEXTCHAR;
    if CHAR ≠ COMMA then
      begin
      if CHAR ≠ SEMI then ERRHEAD
      else NOIT1 ← true
      end
     else CHAR ← NEXTCHAR
    end
  until NOIT1;
  end;
```

effects and
post-conditions: the generic part of the alphabet has been successfully processed.

(r) :1.2.2.4.3:

assumptions: requires ability to invoke ALPHFIN which cmpletes any needed processing after an entire alphabet has been stored

:1.2.2.4.3: ALPHFIN;

effects and **post-conditions:** final processing for an alphabet is completed

The object/assumption table for this expansion is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
(i) :1·2·1:		1	1													1	1																				
(j) :1·2·2:	1	1	1															1	1	1																	
(k) :1·2·2·1:		1																		1																	
(l) :1·2·2·2:																								1													
(m) :1·2·2·3:			1																					1	1	1	1										
(n) :1·2·2·4:	1		1															1	1	1					1	1	1		1	1	1	1		1		1	1
(o) :1·2·2·5:																				1								1									
(p) :1·2·2·4·1:	1		1															1	1	1					1	1	1		1	1	1	1	1	1			
(q) :1·2·2·4·2:			1																						1	1	1		1						1	1	
(r) :1·2·2·4·3:																																					1

1) requires the ability to invoke :3:, ERROR, with a string message
2) requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available
3) NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR
4) an end-of-file condition has occurred
5) requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE
6) requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm
7) CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0
8) write access required for CP
9) C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE
10) requires read access to C
11) requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted
12) "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm

is to be processed
13) requires ability to print the string argument which is passed as the parameter to ERROR
14) ability to perform printing operations
15) requires ability to invoke :1.4: which processes the remaining data images for this algorithm
16) requires write access to PR
17) pr = true means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule
18) MAXA equals the maximum number of alphabets permitted for an algorithm
19) requires read access to MAXA
20) ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR
21) NG contains the number of generic variables encountered for the current algorithm
22) CG[i], 1 ≤ i ≤ NG, equals the i-th generic variable encountered for the current algorithm
23) AG[i], 1 ≤ i ≤ NG, equals the alphabet name which CG[i] is a generic variable
24) a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image
25) read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")"
26) assumes write access to CHAR
27) assumes read access to CHAR
28) requires ability to invoke RULES which names the rule input part, but since RULES is a label in the main program a go to statement can be used
29) ERRHEAD assumes compete control when invoked and handles error messages and further processing
30) NA equals the number of alphabets which have been processed thus far for the current algorithm
31) requires read accesss to NA
32) requires write access to NA
33) requires read/write access to NOIT, which controls a loop that process alphabets
34) requires ability to invoke STORALPH, which stores the alphabet character if all requirements are met,otherwise STORALPH invokes appropriate error routines
35) requires read/write access to NOIT1, which controls a loop that processes generics
36) requires ability to invoke STORGEN which stores the content of CHAR, if legal, otherwise invokes the apporpriate error
37) requires ability to invoke ALPHFIN which cmpletes any needed processing after an entire alphabet has been stored

38) either no alphabets have been processed of all alphabets processed have been correct

This table involves two stages of elaboration. RLB and RUB for the elaboration of :1.2: to :1.2.1: and :1.2.2: are

RUB: ((a) ((e) ((f) ((g , h) ((d) (b , c)) .94) 1.06) .14) .19) 0

RLB: ((a) ((e) ((f) ((g , h) ((d) (b , c)) .94) .99) .07) .10) 0

but the actual expansion leads to

((a) ((e) ((f) ((g , h) ((d) ((c) (i , j) 1.0) .94) .99) .07) .10) 0

Expanding :1.2.2: leads to RLB and RUB

RUB: ((a) ((e) ((f) ((g , h) ((d) ((c) (i , j))

.88) 1.23) .79) .93) .12) .16) 0

RLB: ((a) ((e) ((f) ((g , h) ((d) ((c) (i , j))

0) .88) .86) .93) .04) .05) 0

but the actual entropy loadings are

((a) ((e) ((f) ((g , h) ((d) ((c) ((i) (k , l , m , n , o))

1.33) 1.59) 1.36) 1.48) .12) .16) 0

A better decomposition is

((a) ((e) ((f) ((l) ((k , o) ((m , n) ((i) (d , c, g , h))

1.33) 1.20) 1.33) .54) .36) .12) 0

After examining RLB and RUB for the expansion of :1.2.2.4:, the following good decomposition was found

((a) ((e) ((r) ((f) ((l) ((k , o) ((m , q , p)

((i) ((d , c, g , h)) 1.34) 1.12) 1.26) .49) .09) .12) .16) 0

The major parts in this decomposition are (1) the algorithm interpretation and input parts along with the error processing that can occur there (**c** , **d** , **g** , **h**); (2) alphabet and generic input parts (**m** , **p** , **q**); and (3) the parts that set failure routines.

(s) :11:

assumptions: requires ability to invoke FAIL which correctly determines which objects assume control, a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE, requires read access to C, CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0, read access required for CP, write access required for CP

```
:11: NEXTCHAR: CP ← CP + 1;
if CP > 80 then
FAIL
else
NEXTCHAR ← C[CP];
```

effects and **post-conditions:** the value of :11: is set to be C[CP], i.e. the next available character in the current image, and if no more characters are available, the failure routine is executed

(t) :12:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, NA equals the number of alphabets which have been processed thus far for the current algorithm, requires read accesss to NA, read and write access required for the variable I, requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":", requires ability to invoke TESTGEN which returns only if CHAR is not equal to an already used generic variable for this algorithm, A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters, requires read access for A, AG[i], 1 ≤ i ≤ NG, equals the alphabet name which CG[i] is a generic variable, requires write access to CG, CG[i], 1 ≤ i ≤ NG, equals the i-th generic variable encountered for the current algorithm, NG contains the number of generic variables encountered for the current algorithm, requires read access to NG, requires write access to NG, requires write access to AG, assumes read access to CHAR, CHAR contains the next unstored character from the alphabet or generics being currently processed, AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA, requires read access to AL, requires ability to invoke ERRGEN, which assumes control and invokes an appropriate error routine,

```
:12: STORGEN

TESTLEGAL;
TESTGEN;
for I ← UPA step 1 until AL[0] -1 do
    if CHAR = A[I] then ERRGEN;

NG ← NG + 1;
if NG > MAXG then ERROR ("TOO MANY GENERIC
 VARIABLES");
CG[NG] ← CHAR;
AG[NG] ← NA;
```

effects and **post-conditions**: content of CHAR has been stored as a generic variable with respect to the alphabet currently being processed

(u) :13:

assumptions:

requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":", requires ability to invoke TESTGEN which returns only if CHAR is not equal to an already used generic variable for this algorithm, requires ability to invoke TESTAL which returns only if CHAR is not equal to a character which has already occurred in the alphabet currently being processed, requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA, UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named, requires read access to UPA, requires write access to UPA, A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters, requires write access for A, assumes read access to CHAR, CHAR contains the next unstored character from the alphabet or generics being currently processed

```
:13: STORALPH

TESTLEGAL;
TESTGEN;
TESTAL;

UPA ← UPA - 1;
TEST;
A[UPA] ←CHAR;
```

effects and post-conditions:

the contents of CHAR has been correctly stored as an alphabet character in the current alphabet

(v) :14:

assumptions: AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet, i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA, requires write access to AL, NA equals the number of alphabets which have been processed thus far for the current algorithm, requires read accesss to NA, UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named, requires read access to UPA

```
:14: ALPHFIN

AL[NA] ← UPA;
```

effects and **post-conditions:** lower bound for alphabet NA has been set

(w) :15:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named, requires read access to UPA, LPA names the las cell of an array into which rules are stored, counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named, requires read access to LPA, LPA and UPA are index variables for the same array

```
:15: TEST I.E.
if LPA ≥ UPA then
    ERROR("ALGORITHM TEXT TOO LARGE OR TOO MANY
 ALPHABET
                    CHARACTERS");
```

effects and **post-conditions:** there is an available storage location for a rule rule character or an alphabetic character or a generic

(x) :16:

assumptions: read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", requires the ability to invoke :3:, ERROR, with a string message, assumes read access to CHAR, CHAR contains the next unstored character from the alphabet or generics being currently processed

```
:16:
TESTLEGAL, I.E.

if CHAR = COLON or CHAR = DOT or CHAR = COMMA or
 CHAR = SEMI then
     ERROR("ILLEGAL CHARACTER");
```

effects and **post-conditions:** CHAR is not a colon, period, comma, or semicolon

(y) :17:

assumptions: assumes read access to CHAR, CHAR contains the next unstored character from the alphabet or generics being currently processed, NG contains the number of generic variables encountered for the current algorithm, requires read access to NG, CG[i], $1 \leq i \leq NG$, equals the i-th generic variable encountered for the current algorithm, requires read access to CG, requires ability to invoke ERRGEN, which assumes control and invokes an appropriate error routine, requires read/write access to J

```
:17: TESTGEN I.E.
for J ← 1 step 1 until NG do
     if CHAR = CG[J] then ERRGEN;
```

effects and **post-conditions:** CHAR has not occurred before as a generic variable

(z) :18:

assumptions:

requires read/write access to K, NA equals the number of alphabets which have been processed thus far for the current algorithm, requires read accesss to NA, requires the ability to invoke :3:, ERROR, with a string message, UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named, requires read access to UPA, A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters, requires read access for A, assumes read access to CHAR, CHAR contains the next unstored character from the alphabet or generics being currently processed, requires read access to AL, AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet, i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA

```
:18: TESTAL I.E.
for K ← UPA step 1 until AL[NA] - 1 do
  if CHAR = A[K] then
      ERROR("CHARACTER APPEARS TWICE IN ALPHABET");
```

effects and **post-conditions**:

CHAR has not occurred already in the alphabet being processed

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
(s)	:11:							1	1	1	1										
(t)	:12:	1																			
(u)	:13:																				
(v)	:14:																				
(w)	:15:	1																			
(x)	:16:	1																			
(y)	:17:																				
(z)	:18:	1																			

		21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
(s)	:11:				1															1	1	
(t)	:12:	1	1	1				1			1	1										1
(u)	:13:							1														
(v)	:14:										1	1										
(w)	:15:																					
(x)	:16:					1		1														
(y)	:17:	1	1					1														
(z)	:18:							1			1	1										

		42	43	44	45	46	47	48	49	50	51	52	53	54	56	57	58	59	60	61	62	63	64	65	66	67	68	69
(s)	:11:																											
(t)	:12:	1	1	1	1		1	1	1	1	1	1	1													1		
(u)	:13:	1	1	1						1				1	1	1	1	1		1								
(v)	:14:										1					1	1				1							
(w)	:15:															1	1					1	1	1				
(x)	:16:									1																		
(y)	:17:						1			1			1												1		1	
(z)	:18:			1	1					1	1	1				1	1											1

1) requires the ability to invoke :3:, ERROR, with a string message
2) requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available
3) NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR
4) an end-of-file condition has occurred
5) requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE
6) requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm
7) CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0
8) write access required for CP
9) C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE
10) requires read access to C
11) requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted
12) "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm is to be processed
13) requires ability to print the string argument which is passed as the parameter to ERROR
14) ability to perform printing operations
15) requires ability to invoke :1.4: which processes the remaining

data images for this algorithm
16) requires write access to PR
17) pr = **true** means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule
18) MAXA equals the maximum number of alphabets permitted for an algorithm
19) requires read access to MAXA
20) ability to set the failure routine for NEXTCHAR, i.e. FAIL can be assigned a name which can be invoked if no more characters are available from NEXTCHAR
21) NG contains the number of generic variables encountered for the current algorithm
22) CG[i], 1 ≤ i ≤ NG, equals the i-th generic variable encountered for the current algorithm
23) AG[i], 1 ≤ i ≤ NG, equals the alphabet name which CG[i] is a generic variable
24) a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image
25) read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")"
26) assumes write access to CHAR
27) assumes read access to CHAR
28) requires ability to invoke RULES which names the rule input part, but since RULES is a label in the main program a **go to** statement can be used
29) ERRHEAD assumes compete control when invoked and handles error messages and further processing
30) NA equals the number of alphabets which have been processed thus far for the current algorithm
31) requires read accesss to NA
32) requires write access to NA
33) requires read/write access to NOIT, which controls a loop that process alphabets
34) requires ability to invoke STORALPH, which stores the alphabet character if all requirements are met,otnerwise STORALPH invokes appropriate error routines
35) requires read/write access to NOIT1, which controls a loop that processes generics
36) requires ability to invoke STORGEN which stores the content of CHAP, if legal, otherwise invokes the apporpriate error
37) requires ability to invoke ALPHFIN which cmpletes any needed processing after an entire alphabet has been stored
38) either no alphabets have been processed of all alphabets processed have been correct
39) requires ability to invoke FAIL which correctly determines which objects assume control
40) read access required for CP

41) read and write access required for the variable I
42) requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":"
43) requires ability to invoke TESTGEN which returns only if CHAR is not equal to an already used generic variable for this algorithm
44) A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters
45) requires read access for A
46) NG contains the number of generic variables encountered for the current algorithm
47) requires read access to NG
48) requires write access to NG
49) requires write access to AG
50) CHAR contains the next unstored character from the alphabet or generics being currently processed
51) AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet, i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA
52) requires read access to AL
53) requires ability to invoke ERRGEN, which assumes control and invokes an appropriate error routine
54) requires ability to invoke TESTAL which returns only if CHAR is not equal to a character which has already occurred in the alphabet currently being processed
55) requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA
56) UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named
57) requires read access to UPA
58) requires write access to UPA
59) A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters
60) requires write access for A
61) requires write access to AL
62) LPA names the las cell of an array into which rules are stored counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named
63) requires read access to LPA
64) LPA and UPA are index variables for the same array
65) requires read access to CG
66) requires write access to CG
67) requires read/write access to J
68) requires read/write access to K
69) requires read access for A

After examining entropy loading values for these new objects, the following good decomposition was found

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x) ((c) ((y , u) ((z , t)

(m , q , p)) 1.66) 1.48) 1.44) 1.44) 1.31)

1.00) 1.02) .47) .41) .55) .36) .10) .12) 0

This decomposition is similar to the decomposition at the last stage. Because objects (r) ... (v) were introduced, the generic and alphabet input parts interact strongly with the rest of the program.

Next, :1.3: that inputs rules, is elaborated. A map for this elaboration, as well as a map for several objects invoked by the elaboration of :1.3:, appears below.

(*a) :1.3.1: (236) set failure routine to ERRULENAME.

(*b) :1.3.2: (236) initialize for rule input.

(*c) :1.3.3: (236) get a new input card.

:1.3: — (*d) :1.3.4: (237) as long as the input card is a rule, keep executing :1.3.5: and :1.3.6:.

(*e) :1.3.5: print card; input and store rule.

(*f) :1.3.6: (237) get a new input card.

(*g) :1.3.7: (238) process end of rules conditions.

(*h) :1.3.5.1: (238) print card

(*i) :1.3.5.2: (238) initialize for new rule.

(*j) :1.3.5.3: (238) process rule label.

:1.3.5: — (*k) :1.3.5.4: (239) collect left part of rule.

(*l) :1.3.5.5: (239) initialize for right part.

(*m) :1.3.5.6: (240) collect and store right part.

(*n) :1.3.5.7: (241) collect and store successor label.

(*o) :20: (242) input a label.

(*p) :21: (243) store a label.

(*q) :30: (244) mark rule as a terminal rule.

(*r) :31: (245) process successor part of rule.

(*s) :32: (246) make a character part of this rule.

(*t) :33: (247) initialize for rule processing.

(*u) :34: (248) initialize for processing a new rule.

(*v) :35: (249) initialize for processing right part of rule.

(*w) :36: (250) process end of all rule input conditions.

(*x) :37: (250) check whether the current card should be interpreted as a rule.

(*a) :1.3.1:

assumptions: requires ability to set the failure routine for NEXTCHAR, i.e. the ability to invoke SETFAIL with a variable which names the part which is to be invoked if no more characters are available from NEXTCHAR, ERRULENAME contains the value which indicates a routine which can take control if an error is discovered as rules are being stored, requires read access to ERRULENAME

:1.3.1: SETFAIL(ERRULENAME);

effects and **post-conditions:** failure routine for NEXTCHAR has been set to the part named by ERRULENAME

(*b) :1.3.2:

assumptions: requires ability to invoke INITRA, which initializes the input part for rules for a new algorithm

:1.3.2: EXECUTE ANY NEEDED INITIALIZATIONS FOR INPUTTING THE RULES FOR AN ALGORITHM, I.E. INITRA;

effects and **post-conditions:** rule input part can correctly accept a set of rules for a new algorithm

(*c) :1.3.3:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available

:1.3.3: NEXTCARD;

effects and **post-conditions:** a new card image has been read and is available

(*d) :1.3.4:

assumptions: requires ability to invoke CARDISRULE which determines whether the current card image is to be interpreted as as a rule or not

```
:1.3.4:

while CARDISRULE do
    begin
    :1.3.5: ;
    :1.3.6:
    end;
```

effects and **post-conditions:** all rules for this algorithm have been successfully inputted

(*e) :1.3.5:

assumptions: requires ability to invoke PRINTCARD which prints the current card image,

```
:1.3.5: PRINTCARD;
INPUT AND STORE A RULE;
```

effects and **post-conditions:** card image has been printed and a single rule has been correctly inputted

(*f) :1.3.6:

assumptions: requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available

```
:1.3.6: NEXTCARD;
```

effects and **post-conditions:** a new card image has been inputted and is available

(*g) :1.3.7:

assumptions: requires ability to invoke EDR, end of rules condition processor

```
:1.3.7: EDR;
```

effects and **post-conditions:** all end-of -rule conditions have been correctly resolved

Next, :1.3.5:, which inputs a single rule, is elaborated.

(*h) :1.3.5.1:

assumptions: requires ability to invoke PRINTCARD which prints the current card image

```
:1.3.5.1: PRINTCARD;
```

effects and **post-conditions:** current card image has been printed

(*i) :1.3.5.2:

assumptions: requires ability to invoke INITR which initializes for inputting a new rule

```
:1.3.5.2: INITIALIZE FOR THIS RULE I.E.  INITR;
```

effects and **post-conditions:** rule input is properly initialized to accept a new rule

(*j) :1.3.5.3:

assumptions: requires ability to invoke LABL which collects a label terminated by the character in TERM and leaves the integer label value in LAB, requires ability to invoke STORLABEL which associates the label with the current rule, requires write access to TERM, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")"

```
:1.3.5.3: TERM ← COLON;
          LABL;
          STORLABEL;
```

effects and **post-conditions:** label of current rule has been correctly processed and associated with the current rule

(*k) :1.3.5.4:

assumptions: NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", assumes write access to CHAR, requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":", ability to invoke STORCHAR which returns only if the content of CHAR could be successfully stored with rule being processed

:1.3.5.4: COLLECT AND STORE LEFT PART OF RULE, I.E.

```
CHAR ← NEXTCHAR;
while CHAR ≠ COLON do
    begin
    TESTLEGAL;
    STORCHAR;
    CHAR ← NEXTCHAR
    end;
```

effects and **post-conditions:** the left part of a rule has been correctly stored

(*l) :1.3.5.5:

assumptions: requires ability to invoke INITRIGHT which initializes for processing the input of a right part of a rule

:1.3.5.5: INITIALIZE FOR RIGHT HALF RULE PROCESSING I.E. INITRIGHT

effects and **post-conditions:** the right half of the rule can be correctly processed

(*m) :1.3.5.6:

assumptions: NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, assumes write access to CHAR, assumes read access to CHAR, requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":", read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", ability to invoke STORCHAR which returns only if the content of CHAR could be successfully stored with rule being processed

:1.3.5.6: COLLECT AND STORE RIGHT HALF OF RULE I.E.

```
CHAR ← NEXTCHAR;
while CHAR ≠ DOT ∧ CHAR ≠ SEMI ∧ CHAR ≠ COMMA do
    begin
    TESTLEGAL;
    STORCHAR;
    CHAR ← NEXTCHAR
    end;
```

effects and
post-conditions: right half of the rule has been correctly processed

(*n) :1.3.5.7:

assumptions: read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing, assumes read access to CHAR, a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, assumes write access to CHAR, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, requires ability to invoke PROCTERM which indicates a rule as a terminal rule; requires ability to invoke PROCSUC which processes the successor part of a rule, requires read access to LAB, requires write access to TERM, requires ability to invoke LABL which collects a label terminated by the character in TERM and leaves the integer label value in LAB, a negative value in LAB indicates that no label was collected

:1.3.5.7: COLLECT AND PROCESS SUCCESSOR PART OF RULE, I.E.

```
if CHAR ≠ SEMI then
    begin
    if CHAR = DOT then
        begin
        PROCTERM;
        CHAR ← NEXTCHAR;
        if CHAR ≠ SEMI then
            ERROR("RULE NOT CORRECT")
        end
    else
        if CHAR = COMMA then
            begin
            TERM ← SEMI;
            LABL;
            if LAB < 0 then
               ERROR("SUCCESSOR MISSING")
            PROCSUC;
            end;
        end;
```

effects and post-conditions: the successor part of the current rule has been correctly processed

Below are the elaborations of the additional functions which are needed.

(*o) :20:

assumptions: a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image, TERM contains the non-digit character which is expected to terminate a label, requires read access to TERM, assumes write access to CHAR, assumes read access to CHAR, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, the digits "0", "1", "2", ... , "9" are represented by character codes such that "0" - ZERO = 0, ... ,"9" - ZERO = 9 and the only legal label characters are digits and BLANK, read/write access to LAB is required, legal range of labels is 1 through 100, BLANK contains the representation of a space and ZERO contains the representation of a zero, requires read access to BLANK and ZERO , requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing

```
:20: LABL, I.E.
LAB ← -1;
CHAR ← NEXTCHAR;
while CHAR ≠ TERM do
  begin
  if CHAR ≥ ZERO ∧ CHAR ≤ ZERO + 9 then
    LAB ← (if LAB ≥ 0 then 10 * LAB + CHAR - ZERO
             else CHAR - ZERO)
  else
    if CHAR ≠ BLANK then
      ERROR("ILLEGAL CHARACTER IN LABEL");
    CHAR ← NEXTCHAR
  end;

if LAB = 0 or LAB > 100 then
    ERROR("LABEL OUT OF RANGE");

return the value of LAB;
```

effects and **post-conditions:** a syntactically correct label has been concatenated, if control returns to caller; if no label has been concatenated, the value -1 is returned

(*p) :21:

assumptions: assumes LAB contains a legal label name or -1 which indicates that no label has been concatenated, requires read access to LAB, requires write access to LAB, N1 names the rule being currently inputted, requires read access to N1, requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing, R names a one-dimensional array which contains pointers to rules such that label i names rule R[i]; R[i] = 0 means label i is not defined; R[i] > 0 means the label is defined and R[i] is a pointer to the rule it namws; R[i] < 0 means that label i is undefined but has been referenced by rule -R[i] and is the head of a chain, A names a one-dimensional array which from its lower bound to LPA contains representations of rules, requires write access for A, after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined

```
:21: STORLABEL

if LAB > 0 then
  begin
  if R[LAB] > 0 then ERROR("DOUBLE LABEL
 OCCURRENCE");

if R[LAB] = 0 then R[LAB] ← N1
  else
  while R[LAB] < 0 do
    begin
    TEMP ← - R[LAB];
    R[LAB] ← A[TEMP];
    A[TEMP] ← N1
    end;
  end;
```

effects and **post-conditions:** label LAB has been stored and all undefined references to LAB have been resolved

(*q) :30:

assumptions: after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined, requires read access to N2, requires write access for A, A names a one-dimensional array which from its lower bound to LPA contains representations of rules, a value of -1 in A[N2] indicates that the rule is a terminal rule

```
:30: PROCTERM, I.E.
A[N2] ← -1;
```

effects and **post-conditions:** the rule is marked as terminal

(*r) :31:

assumptions: after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined, R names a one-dimensional array which contains pointers to rules such that label i names rule R[i]; R[i] = 0 means label i is not defined; R[i] > 0 means the label is defined and R[i] is a pointer to the rule it namws; R[i] < 0 means that label i is undefined but has been referenced by rule -R[i] and is the head of a chain, requires write access to R, requires read access to R, A names a one-dimensional array which from its lower bound to LPA contains representations of rules, requires write access for A, assumes LAB contains a legal label name or -1 which indicates that no label has been concatenated, requires read access to N2, requires read access to LAB

:31: PROCSUC, I.E.

```
if R[LAB] > 0 then A[N2] ← R[LAB]
else
    if R[LAB] = 0 then
        begin
        R[LAB] ← -N2;
        A[N2] ← 0
        end
    else
        begin
        A[N2] ← R[LAB];
        R[LAB] ← -N2
        end;
```

effects and
post-conditions: the successor to the current rule has been set or made part of a chain of rules which have the same, yet undefined, successor

(*s) :32:

assumptions: requires read access to NE, NE names a cell in A such that A[NE] indicates the current number of characters in the rule part being processed - left part if the left part is being processed or right part if the right part is being processed, assumes read access to CHAR, NG contains the number of generic variables encountered for the current algorithm, requires read access to NG, CG[i], $1 \le i \le$ NG, equals the i-th generic variable encountered for the current algorithm, requires read access to CG, LPA names the las cell of an array into which rules are stored, counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named, requires read accesss to NA, requires write access to LPA, requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA, requires read/write access to L, A names a one-dimensional array which from its lower bound to LPA contains representations of rules, requires write access for A

:32: STORCHAR I.E.

```
L ← 1;

while l ≤ NG ∧ CG[L] ≠ CHAR do
    L ← L + 1;

LPA ← LPA + 1;
TEST;
if L > NG then
    A[LPA] ← CHAR
else
    A[LPA] ← - L;
A[NE] ← A[NE] + 1;
```

effects and **post-conditions:** A[LPA] is negative if CHAR was a generic and -A[LPA] then indexes the generic in CG; otherwise A[LPA] IS CHAR. A[NE] contains tne number of characters encountered thus far for the rule part being processed

(*t) :33:

assumptions: at most 100 labels are permitted and their definitions appear in the one-dimensional array R[1:100] such that R[i] contains the definition of label i when the label is interpreted as a positive integer, R names a one-dimensional array which contains pointers to rules such that label i names rule R[i]; R[i] = 0 means label i is not defined; R[i] > 0 means the label is defined and R[i] is a pointer to the rule it namws; R[i] < 0 means that label i is undefined but has been referenced by rule -R[i] and is the head of a chain, requires write access to R, NR indicates the rulename which is currently being processed requires write access to NR,

```
:33: INITRA I.E.

NR ← 0;
SETFAIL(ERRULENAME);

for H ← 1 step 1 until 100 do R[H] ← 0;
```

effects and **post-conditions:** the error routine for NEXTCHAR is set to ERRULE and all label definitions are set to 0, i.e. undefined

(*u) :34:

assumptions: NR indicates the rulename which is currently being processed, requires write access to NR, LPA names the las cell of an array into which rules are stored, counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named, requires read accesss to NA, requires write access to LPA, requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA, A names a one-dimensional array which from its lower bound to LPA contains representations of rules, after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined, requires read access to N2, requires write access to N2, NE names a cell in A such that A[NE] indicates the current number of characters in the rule part being processed - left part if the left part is being processed or right part if the right part is being processed, requires read access to NE, requires write access to NE

:34: INITR - initialize for new rule i.e.

```
NR ← NR + 1;
LPA ← LPA + 4;
TEST;
A[LPA - 3] ← NR;
if NR > 1 then A[N1] ← LPA - 2;
N1 ← LPA - 2;
A[N1] ← -1;
N2 ← LPA - 1;
A[N2] ← 2;
NE ← LPA;
A[NE] ← 0;
```

effects and
post-conditions: after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined

(*v) :35:

assumptions:

after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined, LPA names the las cell of an array into which rules are stored, counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named, requires read accesss to NA, requires write access to LPA, requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA, NE names a cell in A such that A[NE] indicates the current number of characters in the rule part being processed - left part if the left part is being processed or right part if the right part is being processed, requires read access to NE, requires write access to NE

:35: INITRIGHT I.E.

```
LPA ← LPA + 1;
TEST;
NE ← LPA;
A[NE] ← 0;
```

effects and
post-conditions:

A[NE] = 0 and indicates the current number of characters in the right half of the current rule

(*w) :36:

assumptions:

R names a one-dimensional array which contains pointers to rules such that label i names rule R[i]; R[i] = 0 means label i is not defined; R[i] > 0 means the label is defined and R[i] is a pointer to the rule it namws; R[i] < 0 means that label i is undefined but has been referenced by rule -R[i] and is the head of a chain, requires read access to R, NR indicates the rulename which is currently being processed, requires read access to NR, requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing, requires read/write access to Q :36: EDR I.E.

```
for  Q ← 1 step 1 until 100 do
    if R[Q] < 0 then ERROR("UNDEFINED LABEL(S)");

if NR < 0 then ERROR("ALGORITHM CONTAINS NO RULES");
```

effects and **post-conditions:**

a set of rules has successfully been inputted and constitutes a syntactically correct algorithm

(*x) :37:

assumptions:

read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")", C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE, requires read access to C, a colon in column 4 when processing rules indicates that the card is to be interpreted as a rule

:37: CARDISRULE I.E.

C[4] = COLON

effects and **post-conditions:**

true if card is to be interpreted as a rule; **false** otherwise

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
(*a) :1·3·1:																				1											
(*b) :1·3·2:																															
(*c) :1·3·3:		1																													
(*d) :1·3·4:																															
(*e) :1·3·5:																															
(*f) :1·3·6:		1																													
(*g) :1·3·7:																															
(*h) :1·3·5·1:																															
(*i) :1·3·5·2:																															
(*j) :1·3·5·3:																									1						
(*k) :1·3·5·4:			1																					1	1	1					
(*l) :1·3·5·5:																															
(*m) :1·3·5·6:			1																					1	1	1	1				
(*n) :1·3·5·7:	1		1																					1	1	1	1				
(*o) :20:	1		1																					1		1	1				
(*p) :21:	1																														
(*q) :30:																															
(*r) :31:																															
(*s) :32:																					1	1					1				1
(*t) :33:																															
(*u) :34:																															1
(*v) :35:																															1
(*w) :36:	1																														
(*x) :37:									1	1															1						

	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
(*a) :1·3·1:																												
(*b) :1·3·2:																												
(*c) :1·3·3:																												
(*d) :1·3·4:																												
(*e) :1·3·5:																												
(*f) :1·3·6:																												
(*g) :1·3·7:																												
(*h) :1·3·5·1:																												
(*i) :1·3·5·2:																												
(*j) :1·3·5·3:																												
(*k) :1·3·5·4:	1																											
(*l) :1·3·5·5:																												
(*m) :1·3·5·6:	1																											
(*n) :1·3·5·7:																												
(*o) :20:																												
(*p) :21:																			1									
(*q) :30:																			1									
(*r) :31:																			1									
(*s) :32:						1								1					1		1			1				
(*t) :33:																												
(*u) :34:														1							1							
(*v) :35:														1							1							
(*w) :36:																												
(*x) :37:																												

```
                    70717273747576777879808182838485868788899091
(*a) :1·3·1:
(*b) :1·3·2:         I
(*c) :1·3·3:
(*d) :1·3·4:           I
(*e) :1·3·5:             I
(*f) :1·3·6:
(*g) :1·3·7:               I
(*h) :1·3·5·1:           I
(*i) :1·3·5·2:           I
(*j) :1·3·5·3:                   I I I
(*k) :1·3·5·4:                         I
(*l) :1·3·5·5:                           I
(*m) :1·3·5·6:                         I
(*n) :1·3·5·7:                   I   I     I I I
(*o) :20:
(*p) :21:                                      I   I I     I
(*q) :30:                                          I       I
(*r) :31:                                      I   I I I I I
(*s) :32:                                                  I I I I
(*t) :33:                                            I I           I
(*u) :34:                                          I       I I I I
(*v) :35:                                          I         I I I
(*w) :36:                                            I   I
(*x) :37:

                    92939495969798990001020304050607080910111213141516171819
(*a) :1·3·1:                                         I I
(*b) :1·3·2:
(*c) :1·3·3:
(*d) :1·3·4:
(*e) :1·3·5:
(*f) :1·3·6:
(*g) :1·3·7:
(*h) :1·3·5·1:
(*i) :1·3·5·2:
(*j) :1·3·5·3:
(*k) :1·3·5·4:
(*l) :1·3·5·5:
(*m) :1·3·5·6:
(*n) :1·3·5·7:
(*o) :20:                    I I I I I I I
(*p) :21:                                    I I I I
(*q) :30:                                                I         I
(*r) :31:                                    I           I
(*s) :32:                                                  I
(*t) :33:            I   I
(*u) :34:            I   I I                             I   I
(*v) :35:                  I
(*w) :36:            I I                                         I
```

(*x) :37: 1

1) requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing
2) requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available
3) NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR
4) an end-of-file condition has occurred
5) requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE
6) requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm
7) CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0
8) write access required for CP
9) C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result of the last execution of GETIMAGE
10) requires read access to C
11) requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted
12) "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm is to be processed
13) requires ability to print the string argument which is passed as the parameter to ERROR
14) ability to perform printing operations
15) requires ability to invoke :1.4: which processes the remaining data images for this algorithm
16) requires write access to PR
17) pr = true means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule
18) MAXA equals the maximum number of alphabets permitted for an algorithm
19) requires read access to MAXA
20) requires ability to set the failure routine for NEXTCHAR, i.e. the ability to invoke SETFAIL with a variable which names the part which is to be invoked if no more characters are available from NEXTCHAR
21) NG contains the number of generic variables encountered for the current algorithm
22) CG[i], $1 \leq i \leq$ NG, equals the i-th generic variable encountered for the current algorithm
23) AG[i], $1 \leq i \leq$ NG, equals the alphabet name which CG[i] is a generic variable

24) a failure routine has been set if NEXTCHAR cannot
provide additional characters from the current image
25) read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which
contain the values of ".", ",", ":", ";", "(", ")"
26) assumes write access to CHAR
27) assumes read access to CHAR
28) requires ability to invoke RULES which names the rule input
part, but since RULES is a label in the main program
a go to statement can be used
29) ERRHEAD assumes compete control when invoked
and handles error messages and further processing
30) NA equals the number of alphabets which have been processed
thus far for the current algorithm
31) requires read accesss to NA
32) requires write access to NA
33) requires read/write access to NOIT, which controls a
loop that process alphabets
34) requires ability to invoke STORALPH, which stores the alphabet
character if all requirements are met,otherwise STORALPH
invokes appropriate error routines
35) requires read/write access to NOIT1, which controls a loop
that processes generics
36) requires ability to invoke STORGEN which stores the content
of CHAR, if legal, otherwise invokes the apporpriate
erroi
37) requires ability to invoke ALPHFIN which cmpletes any
needed processing after an entire alphabet has been
stored
38) either no alphabets have been processed of all alphabets
processed have been correct
39) requires ability to invoke FAIL which correctly determines
which objects assume control
40) read access required for CP
41) read and write access required for the variable I
42) requires ability to invoke TESTLEGAL which returns only if
CHAR is not ".", ",", ";", ":"
43) requires ability to invoke TESTGEN which returns only if
CHAR is not equal to an already
used generic variable for this algorithm
44) A names a one-dimensional array, which from UPA to its
upper bound contains alphabetic characters
45) requires read access for A
46) NG contains the number of generic variables encountered
for the current algorithm
47) requires read access to NG
48) requires write access to NG
49) requires write access to AG
50) CHAR contains the next unstored character
from the alphabet or generics being currently processed
51) AL[1] ... AL[NA] names the index of the lower bound of the

characters in an alphabet, i.e. AL[i] is the lower
bound for the i-th alphabet and AL[i-1] is the
upper bound for that alphabet, where AL[0] equals the
initial value plus 1 of UPA

52) requires read access to AL
53) requires ability to invoke ERRGEN, which assumes
control and invokes an appropriate error routine
54) requires ability to invoke TESTAL which returns only if
CHAR is not equal to a character which has already occurred in the
alphabet currently being processed
55) requires ability to invoke TEST which returns only if there
storage space as indicated by the values of LPA and UPA
56) UPA names the last cell of an array into which an alphabetic
character was stored, counting from the top of some
one-dimensional array. UPA is decremented by 1
each time an available cell needs to be named
57) requires read access to UPA
58) requires write access to UPA
59) A names a one-dimensional array, which from UPA to its
upper bound contains alphabetic characters
60) requires write access for A
61) requires write access to AL
62) LPA names the las cell of an array into which rules
are stored, counting from the lower bound of a
one-dimensional array . LPA is incremented by one each
time an available cell needs to be named
63) requires read access to LPA
64) LPA and UPA are index variables for the same array
65) requires read access to CG
66) requires write access to CG
67) requires read/write access to J
68) requires read/write access to K
69) requires read access for A
70) requires ability to invoke INITRA, which initializes the
input part for rules for a new algorithm
71) requires ability to invoke CARDISRULE which determines
whether the current card image is to be interpreted as
as a rule or not
72) requires ability to invoke PRINTCARD
which prints the current card image
73) requires ability to invoke EDR, end of rules condition
processor
74) requires ability to invoke INITR which initializes
for inputting a new rule
75) requires ability to invoke LABL which collects a label
terminated by the character in TERM and leaves the integer
label value in LAB
76) requires ability to invoke STORLABEL which associates
the label with the current rule
77) requires write access to TERM

78) ability to invoke STORCHAR which returns only if the content of CHAR could be successfully stored with rule being processed
79) requires ability to invoke INITRIGHT which initializes for processing the input of a right part of a rule
80) requires ability to invoke PROCTERM which indicates a rule as a terminal rule; requires ability to invoke PROCSUC which processes the successor part of a rule
81) requires read access to LAB
82) a negative value in LAB indicates that no label was collected
83) after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined
84) R names a one-dimensional array which contains pointers to rules such that label i names rule R[i];
    R[i] = 0 means label i is not defined;
    R[i] > 0 means the label is defined and R[i] is a pointer to the rule it namws;
    R[i] < 0 means that label i is undefined but has been referenced by rule -R[i] and is the head of a chain
85) requires write access to R
86) requires read access to R
87) A names a one-dimensional array which from its lower bound to LPA contains representations of rules
88) requires read access to NE
89) NE names a cell in A such that A[NE] indicates the current number of characters in the rule part being processed - left part if the left part is being processed or right part if the right part is being processed
90) requires write access to LPA
91) at most 100 labels are permitted and their definitions appear in the one-dimensional array R[1:100] such that R[i] contains the definition of label i when the label is interpreted as a positive integer
92) NR indicates the rulename which is currently being processed
93) requires read access to NR
94) requires write access to NR
95) requires write access to NE
96) TERM contains the non-digit character which is expected to terminate a label
97) requires read access to TERM
98) the digits "0", "1", "2", ... , "9" are represented by character codes such that "0" - ZERO = 0, ... ,"9" - ZERO = 9

and the only legal label characters are digits and BLANK
99) read/write access to LAB is required
100) legal range of labels is 1 through 100
101) BLANK contains the representation of a space and ZERO contains the representation of a zero
102) requires read access to BLANK and ZERO
104) assumes LAB contains a legal label name or -1 which indicates that no label has been concatenated
105) requires write access to LAB
106) N1 names the rule being currently inputted
107) requires read access to N1
108) ERRULENAME contains the value which indicates a routine which can take control if an error is discovered as rules are being stored
109) requires read access to ERRULENAME
110) requires read access to N2
111) requires read/write access to L
112) requires write access to N2
113) a colon in column 4 when processing rules indicates that the card is to be interpreted as a rule
114) requires read/write access to Q
115) a value of -1 in A[N2] indicates that the rule is a terminal rule
116) ERRHEADNAME is a variable which names the routine ERRHEAD
117) requires read access to ERRHEADNAME
118) RULESNAME is a variable which names the routine RULES which inputs the rules for an algorithm
119) requires read access to RULESNAME

RUB for the expansion of :1.3: is

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x) ((c) ((y , u) ((z , t)

(m , q , p)) 1.73) 1.28) 1.55) 1.50) 1.39) 1.11)

1.12) .39) .35) .47) .30) .08) .10) 0

The actual loadings are

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x)

((*a , *b , *c , *d , *e , *f , *g) ((y , u) ((z , t)

(m , q , p)) 1.29) 1.28) .83) 1.16) 1.02) .96) .98) .54)

.35) .47) .30) .07) .10) 0

These values are mostly lower than the RUB for the expansion. Consequently, no attempt will be made to find a better decomposition for this stage.

RUB for the expansion of :1.3.5: is

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x)

((*a , *b , *c , *d , *e , *f , *g) ((y , u) ((z , t)

(m , q , p)) 1.03) 1.06) .83) .91) .65) .87) .88) .48

.31) .42) .26) .08) .09) 0

Unfortunately, the actual loadings are

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x)

((*a , *b , *c , *d , *h , *i , *j , *k , *l ,

*m , *n , *f , *g) ((y , u) ((z , t)

(m , q , p)) 1.29) 1.28) .83) 1.16) 1.02) .96) .98) .54)

.35) .47) .30) .07) .10

A modification of this decomposition leads to the following better decomposition

((a) ((r) ((e) ((f) ((s) ((l) ((k , o) ((i)

((d , g , h) ((v , w) ((x)

((*a , *b , *c , *d , *h , *i , *j , *l ,

*f , *g) ((y , u) ((z , t)

(m , q , p , *k , *m , *n)) 1.57) 1.41) 1.39) 1.36)

.97) 1.00) 1.01) .47) .49) .60) .26) .07) .09) 0

Here, those objects that recognize and store information for the rule execution part (m , q , *k , *m , *n) interact most with the rest of the program. Also, those objects that control and invoke the above mentioned objects share much information and appear grouped together (*a , *b, *c , *d , *h , *i , *j , *l , *g , *f). Since the remaining objects in the table will be used - in part - by the expansion of :1.4:, an analysis of the entropy loadings involving them is postponed until after the elaboration of :1.4:.

Below is a map of the elaboration of :1.4: and most of the remaining objects.

:1.4:

(>a) :1.4.1: (262) control input of the initial register contents and markov algorithm execution.

(>b) :1.4.2: (262) print card; input initial register contents.

(>c) :1.4.3.1: (263) initialize for execution of currently stored algorithm and the current register contents.

(>d) :1.4.3.2: (264) control attempts to apply the rules.

(>e) :1.4.3.3: (264) attempt to apply current rule.

(>f) :1.4.3.4: (265) replace matched string and possibly print the register contents; set next rule to the successor for a successful application of the current rule.

(>g) :1.4.3.5: (265) set next rule to be the successor for a failure to match left part of current rule.

(>h) :1.4.3.6: (266) terminate algorithm interpretation if necessary.

(>i) :40: (267) search for an instance of the left half of a rule in the register.

(>j) :41: (269) replace matched string by the right half of the current rule.

(>k) :42: (270) adjust the register to accomodate the right half of the current rule.

(>l) :43: (271) insert the right half of the current rule into the register.

(>m) :44: (272) test whether a generic variable matches a character in the register.

(>n) :45: (272) test whether a rule character is a generic.

(>o) :46: (273) get a character from the left part of a rule.

(>p) :47: (273) get the successor for a rule that corresponds to a successful application of the rule.

(>q) :48: (273) get the successor for a rule that corresponds to an unsuccessful application of the rule.

(>r) :49: (274) get the location of the first character of the left part of the current rule.

(>s) :50: (274) indicate an error in the heading for an algorithm.

(>t) :51: (275) invoke the rule input part of the program.

(>u) :52: (274) indicate an error in a rule.

(>v) :53: (275) indicate an error in an initial register data image

(>w) :54: (275) indicate a generic variable occurring in the data.

(>x) :55: (275) initialize all generic variables to be undefined.

(>y) :65: (276) get character from the right part of the current rule.

(>z) :66: (276) test whether a generic variable is defined.

(<a) :67: (276) get the length of the right part of the current rule.

(<b) :68: (277) get the character associated with a generic variable.

(<c) :70: (277) get the length of the left part of the current rule.

Below is an elaboration of :1.4:, which executes an algorithm with respect to its data images

(>a) :1.4.1:

assumptions: assumes all rules have been inputted correctly; assumes that the current card isrepresents an initial register contents; assumes that NEXTCARD returns control only if a data image is available for the current algorithm

```
:1.4.1: repeat
    begin
    :1.4.2: ;
    :1.4.3: ;
    :1.4.4:
    end
until false;
```

(>b) :1.4.2:

assumptions: requires the ability to invoke :4:, ERRORE, with a string message, which does not return but handles further processing at the execution stage, requires ability to invoke PRINTCARD which prints the current card image, ERRDATANAME contains a value which indicates a routine which indicates a routine which can take control if an error is discovered while an initial register contents is being input, requires read access to ERRDATANAME, requires ability to set the failure routine for NEXTCHAR, i.e. the ability to invoke SETFAIL with a variable which names the part which is to be invoked if no more characters are available from NEXTCHAR, REG, is a one-dimensional array which contains the characters in the register, assumes write access to REG, assumes read access to MAXRL which contains the maximum number of characters permitted in the register, NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR, requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":", requires ability to invoke TESTGEN which returns only if CHAR is not equal to an already used generic variable for this algorithm, requires write access to RL, requires read/write access to RPP which is used as a temporary register position pointer when the register is being initially filled, requires read/write access to RC which is used to contain single characters from the current data card when the register is being filled, read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")"

:1.4.2: PRINTCARD; INPUT THE INITIAL REGISTER CONTENTS, I.E.

```
PRINTCARD;
SETFAIL(ERRDATANAME);
RPP ← 0;

RC ← NEXTCHAR;
while RC ≠ SEMI do
    begin
    RPP ← RPP + 1;
    if RPP > MAXRL then
        ERRORE("REGISTER OVERFLOW");
    TESTLEGAL;
    TESTGEN;
    REG[RPP] ← RC;
    RC ← NEXTCHAR;
    end;

RL ← RPP;
```

effects and **post-conditions:** an initial register contents has been correctly set into the register and RL contains the number of characters in the register

(>c) :1.4.3.1:

assumptions: the name of the first rule is 2, MAXT1 contains the number of trial rule applications still permitted for this execution of an algorithm, requires write access to MAXT1, write access to RULENAME required, RULENAME names the rule currently being processed

:1.4.3.1: INITIALIZE PROCESSING FOR CURRENT ALGORITHM WITH CURRENT REGISTER CONTENTS, I.E.

```
RULENAME ← 2;
MAXT1 ← MAXT;
```

effects and **post-conditions:** processing initialized for current algorithm and register contents

(>d) :1.4.3.2:

assumptions: RULENAME names the rule currently being processed, requires read access to TRM, FIN is a result of NOSUC which indicates that there are no more rules which can be applied, requires read access to FIN, read access to RULENAME required, TRM is a result of SUCSUC and UNSUCSUC which indicates that the algorithm should terminate, i.e. TERM names no legal and indicates termination

```
:1.4.3.2: while RULENAME ≠ TERM ∧ RULENAME ≠ FIN do
begin
:1.4.3.3: ;
:1.4.3.4: ;
:1.4.3.5: ;
:1.4.3.6: ;
end;
```

effects and
post-conditions: an algorithm has been executed with respect to an initial register contents

(>e) :1.4.3.3:

assumptions: requires ability to invoke SEARCH which searches for a match of the left part of rule RULENAME, and which returns the value **true** if a match is found, **false** otherwise, requires the ability to invoke :4:, ERRORE, with a string message, which does not return but handles further processing at the execution stage, MAXT1 contains the number of trial rule applications still permitted for this execution of an algorithm, requires read access to MAXT1, requires write access to MAXT1

```
:1.4.3.3:

MAXT1 ← MAXT1 - 1;
if MAXT1 < 0 then ERRORE("MAXIMUM NUMBER OF TRIALS EXCEEDED");
if  SEARCH then
    :1.4.3.4: else :1.4.3.5: ;
```

effects and
post-conditions: the rule, RULENAME, has been applied, if possible, to the register contents and a new successor rule has been set in RULENAME

(>f) :1.4.3.4:

assumptions: RULENAME names the rule currently being processed, requires ability to invoke REPLACE, which has the effect of replacing the register contents with the right part of the rule named by RULENAME, where the left part was matched, requires the ability to invoke SUCSUC which has the value of the success to rule RULENAME for a successful application of rule RULENAME, write access to RULENAME required, requires read access to PR, pr = **true** means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule, requires the ability to invoke PRINTREG which prints the contents of the register

```
:1.4.3.4: REPLACE;
          RULENAME ← SUCSUC;
          if PR then PRINTREG;
```

effects and **post-conditions:** the right part of rule, RULENAME, has replaced the left-most occurrence of the left part of the rule, RULENAME

(>g) :1.4.3.5:

assumptions: RULENAME names the rule currently being processed, requires the ability to invoke UNSUCSUC which has the value of the name of the successof to rule RULENAME after an unsuccessful application of rule RULENAME, write access to RULENAME required,

```
:1.4.3.5:  RULENAME ← UNSUCSUC;
```

effects and **post-conditions:** RULENAME contains the name of the rule which is to be attempted next if the current rule could not be applied successfully

(>h) :1.4.3.6:

assumptions: RULENAME names the rule currently being processed, requires the ability to invoke :4:, ERRORE, with a string message, which does not return but handles further processing at the execution stage, requires the ability to invoke PRINTREG which prints the contents of the register, pr = **true** means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule, requires read access to PR, if RULENAME = TRM then dot termination has occurred, otherwise the rules have been exhausted, read access to RULENAME required, TRM is a result of SUCSUC and UNSUCSUC which indicates that the algorithm should terminate, i.e. TERM names no legal and indicates termination

```
:1.4.3.6: if RULENAME = TRM then
              begin
              if NOT(PR) then
                PRINTREG;
                ERRORE("DOT TERMINATION")
              end
              else
                ERRORE("RULES EXHAUSTED");
```

effects and
post-conditions: The appropriate termination message has been printed and ERRORE has been invoked to handle further processing

(>i) :40:

assumptions: requires the ability to invoke LEFT which names the location which is the first character of the left part of rule, RULENAME, requires ability to invoke LENGTHL which has the value of the number of characters in the left part of the rule named by its parameter, RULENAME names the rule currently being processed, read access to RULENAME required, LEN names the number of the character in the left part of the rule currently being processed, requires read access to LEN, requires write access to LEN, LFT names the starting location for the left part of the current rule name, requires read access to LFT, requires write access to LFT, requires read/write access to NOSUC, RL contains the current register length, requires read access to RL, requires ability to invoke RULLFTCHR(A,B) which produces the value of the B-th character of the left part which starts at A, and requires ability to invoke INITGEN which sets the definitions of all generic variables to be undefined, RP names the character position where a character sequence is to be replaced, requires read access to RP, requires write access to RP, requires ability to invoke GENERIC which has the value **true** if its parameter represents a generic variable, requires read/write access to CC, requires read/write access to LHP, REG, is a one-dimensional array which contains the characters in the register, assumes read access to REG

:40: SEARCH I.E.

```
LFT ← LEFT(RULENAME);
LEN ← LENGTHL(RULENAME);
NOSUC ← true;
if RL ≥ LEN then
    begin
    RP ← 0;
    while RP < RL + 1 - LEN ∧ NOSUC do
        begin
        RP ← RP + 1
        LHP ← 1;
        NOSUC ← false;
        INITGEN;
        while LHP ≤ LEN ∧ NOT(NOSUC) do
            begin
            CC ← RULLFTCHR(LFT, LHP);
```

```
                if GENERIC(CC) then
                    begin
                    if NOT(MATCHGEN(CC, REG[RP + LHP - 1] ) then
                        NOSUC ← true
                    end
                else
                    if CC ≠ REG[RP + LHP - 1] then
                        NOSUC ← true
                LHP ← LHP + 1
                end;
            end;
        end;
    SEARCH ← NOT(NOSUC);
```

effects and post-conditions: SEARCH =true if the left part of rule RULENAME matches a substring of the register and sets up internal variables which can be used by REPLACE for replacing the first occurrence of a match from the left end of the register by the right part of rule RULENAME; **false** otherwise

(>j) :41:

assumptions: requires ability to invoke LENGTHL which has the value of the number of characters in the left part of the rule named by its parameter, requires the ability to invoke :4:, ERRORE, with a string message, which does not return but handles further processing at the execution stage, requires ability to invoke ADJUST whose first parameter indicates the number of characters in the left part of a rule and whose second parameter indicates the number of characters in the right part of the rule. ADJUST modifies the register, if necessary, so that the right part can be inserted where the matched left part is, requires ability to invoke INSERT which inserts the appropriate right part over the matched left part, requires ability to invoke LENGTHR which returns the value of the number of characters in the right part of the rule named by its parameter, requires read/write access to LEN and LENR, requires read access to LENR, requires write access to LENR

:41: REPLACE I. E.

```
LENR ← LENGTHR(RULENAME);
LENL ← LENGTHL(RULENAME);
if RL - LENL + LENR > MAXRL then
    ERRORE("REGISTER OVERFLOW");
ADJUST(LENL,LENR);
INSERT;
```

effects and **post-conditions:** the right part of rule, RULENAME, has replaced the leftmost occurrence of the left part of RULEMAME

(>k) :42:

assumptions: RL contains the current register length, requires read access to RL, requires write access to RL, requires read access to L1, L2, and TP; L1 contains the length of a character sequence being replaced by a character sequence of length L2, REG, is a one-dimensional array which contains the characters in the register, assumes read access to REG, assumes write access to REG, RP names the character position where a character sequence is to be replaced, requires read access to RP, assumes L1 ≠ L2

```
:42: ADJUST(L1,L2); I.E.

if L1 < L2 then
    begin
    for TP ← RL step -1 until RP + L1 do
        REG[TP + L2 - L1] ← REG[TP];
    end
else
    begin
    for TP ← RP + L1 step 1 until RL do
        REG[TP + L2 - L1] ← REG[TP];
    end;

    RL ← RL + N2 - N1;
```

effects and **post-conditions:** the register is modified so that a replacement of a string starting at RP of length L1 can take place correctly for a string of length L2

(>I) :43:

assumptions: requires read write access to TQ and CQ, RP names the character position where a character sequence is to be replaced, requires read access to RP, requires ability to invoke GENERIC which has the value **true** if its parameter represents a generic variable, requires ability to invoke RULRTCHAR(A) which returns the character which is the A-th character of the right side of rule RULENAME, requires the ability to invoke UNDEF(A) which returns the value **true** if A is an undefineu generic variable representation for this rule application; **false** otherwise, requires ability to invoke VALG(A) which produces the character associated with the generic representation A for this rule application, requires the ability to invoke :4:, ERRORE, with a string message, which does not return but handles further processing at the execution stage, REG, is a one-dimensional array which contains the characters in the register, assumes write access to REG, assumes LENR contains the length of the string being inserted, requires read access to LENR

```
:43: INSERT I.E.
for TQ ← 1 step 1 until LENR do
    begin
    CQ ← RULRTCHAR(TQ);
    if GENERIC(CQ) then
        begin
        if UNDEF(CQ) then
            ERRORE("UNDEFINED GENERIC USED IN RIGHT PART OF RULE")
        else REG[RP + TQ - 1] ← VALG(CQ)
        end
    else
         REG[RP + TQ - 1] ← CQ
    end;
```

effects and **post-conditions:** the characters in the right part of a rule have been stored into successive locations of REG starting at RP

(>m) :44:

assumptions: AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet, i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA, requires read access to AL, AG[i], 1 ≤ i ≤ NG, equals the alphabet name which CG[i] is a generic variable, requires read access to AG, G is a one-dimensional array such that G[i] = -1 if generic variable i is not defined after a successful search for a left part of a rulei, otherwise G[i] > 0 and is the character corresponding to the generic i, requires read access to G, requires write access to G, requires read/write access to NMAT

:44: MATCHGEN(CC,Q) I.E. NMAT ← **true**; **if** G[-CC] < 0 **then begin** T ← AL[AG[-CC]]; **while** NMAT ∧ T < AL[AG[-CC] -1] **do begin if** Q = A[T] **then begin** G[-CC] ← A[T]; NMAT ← **false end**; **end else if** G[-CC] = Q **then** NMAT ← **false**

MATCHGEN ← NOT(NMAT);

effects and **post-conditions:** MATCHGEN = **true** if the generic indicated by CC is matched by Q; **false** otherwise

(>n) :45:

assumptions: assumes that a generic in a rule has been saved as a negative value

:45: GENERIC(S) I.E.

GENERIC ← S < 0;

effects and **post-conditions:** GENERIC = **true** if S represents a generic variable; **false** otherwise

(>o) :46:

assumptions: requires read access for A, assumes Q is a legal character pointer into the left

:46: RULLFTCHR(P,Q) I.E.

RULCHAR ← A[P + Q + 2];

effects and
post-conditions: RULLFTCHR = the Q-th character in the left part of the rule named by P

(>p) :47:

assumptions: requires read access for A, assumes A[RULENAME + 1] names the successor for rule RULENAME for a successful application of rule RULENAME, RULENAME names the rule currently being processed, read access to RULENAME required,

:47: SUCSUC ← A[RULENAME + 1] ;

effects and
post-conditions: SUCSUC names the rule to be tried next after a successful application of rule, RULENAME

(>q) :48:

assumptions: requires read access for A, RULENAME names the rule currently being processed, read access to RULENAME required, assumes A[RULENAME] names the successor rule for rule RULENAME after an unsuccessful application of rule RULENAME

:48: UNSUCSUC ← A[RULENAME];

effects and
post-conditions: UNSUCSUC names the rule to be tried next after an unsuccessful application fo rule RULENAME

(>r) :49:

assumptions: assumes X + 2 is an index which names the first character of the left part of a rule -1

:49: LEFT(X) I.E.
LEFT ← X + 2;

effects and
post-conditions: LEFT is a value which names the left part of rule X

(>s) :50:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing

:50: ERRHEAD, I.E.

ERROR("HEADING NOT CORRECT");

effects and
post-conditions: :3:, ERROR, has been invoked and handles all further processing

(>t) :51:

assumptions: requires ability to invoke RULES which names the rule input part, but since RULES is a label in the main program a **go to** statement can be used

:51: **go to** RULES;

effects and
post-conditions: the rule input part of the program is given control for further processing

(>u) :52:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing

:52: ERRULE I. E.

ERROR("RULE NOT CORRECT");

effects and
post-conditions: :3:, ERROR, has been invoked and handles further processing

(>v) :53:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing

:53: ERRDATA I.E.

```
ERRORE("DATA NOT CORRECT");
```

effects and **post-conditions:** *3:, ERROR, has been invoked and handles further processing

(>w) .54:

assumptions: requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing

:54: ERRGEN I.E.

```
ERROR("GENERIC VARIABLE IN DATA");
```

effects and **post-conditions:** :3:, ERROR, has been invoked and handles further processing

(>x) :55:

assumptions: requires read/write access to QK, G is a one-dimensional array such that G[i] = -1 if generic variable i is not defined after a successful search for a left part of a rulei, otherwise G[i] > 0 and is the character corresponding to the generic i, requires write access to G, NG contains the number of generic variables encountered for the current algorithm, requires read access to NG

:55: INITGEN I.E.

```
for QK ← 1 step 1 until NG do
G[QK] ← -1;
```

effects and **post-conditions:** the definitions of the generic variables have been set to undefined

(>y) :65:

assumptions: requires read access for A, assumes Y is a legal character pointer into the right part of a rule whose first character is named by X + LENGTHL(X) + 4, requires ability to invoke LENGTHL which has the value of the number of characters in the left part of the rule named by its parameter

:65: RULRTCHR(X,Y) I.E.
RULRTCHR ← A[X + LENGTHL(X) + 3 + Y] ;

effects and
post-conditions: RULRTCHR is the Y-th character in the right part of rule X

(>z) :66:

assumptions: G is a one-dimensional array such that G[i] = -1 if generic variable i is not defined after a successful search for a left part of a rulei, otherwise G[i] > 0 and is the character corresponding to the generic i, requires read access to G

:66: UNDEF(X) I.E. UNDEF ← G[-X] > 0;

effects and
post-conditions: UNDEF = **true** if no definition for the generic indicated by X exists; otherwise **false**

(<a) :67:

assumptions: requires read access for A, requires ability to invoke LENGTHL which has the value of the number of characters in the left part of the rule named by its parameter, assumes A[X + LENGTHL(X) + 3] is the number of characters in the right part of rule X

:67: LENGTHR(X) I.E.

LENGTHR ← A[X + LENGTHL(X) + 3];

effects and
post-conditions: LENGTHR equals the length of the right part of the rule indicated by X

(<b) :68:

assumptions: G is a one-dimensional array such that G[i] = -1 if generic variable i is not defined after a successful search for a left part of a rulei, otherwise G[i] > 0 and is the character corresponding to the generic i, requires read access to G

:68: VALG(X) I.E.

VALG ← G[-X];

effects and **post-conditions:** VALG equals the character associated with the generic variable indicated by X

(<c) :70:

assumptions: requires read access for A, assumes read access to X and assumes that the length of the left part of a rule named by X is contained in A[X + 2]

:70: LENGTHL(X) I.E.

LENGTHL ← A[X + 2];

effects and **post-conditions:** LENGTHL is the length in characters of the left part of the rule named by X

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
>a	:1·4·1:																									
>b	:1·4·2:			1																	1					1
>c	:1·4·3·1:																									
>d	:1·4·3·2:																									
>e	:1·4·3·3:																									
>f	:1·4·3·4:																	1								
>g	:1·4·3·5:																									
>h	:1·4·3·6:																	1								
>i	:40:																									
>j	:41:																									
>k	:42:																									
>l	:43:																									
>m	:44:																							1		
>n	:45:																									
>o	:46:																									
>p	:47:																									
>q	:48:																									
>r	:49:																									
>s	:50:	1																								
>t	:51:																									
>u	:52:	1																								
>v	:53:	1																								
>w	:54:	1																								
>x	:55:																									
>y	:65:																									
>z	:66:																									
<a	:67:																									
<b	:68:																									
<c	:70:																									

```
                        28293031323334353637383940414243444546474849505152
>a  :1.4.1:
>b  :1.4.2:                                          1 1
>c  :1.4.3.1:
>d  :1.4.3.2:
>e  :1.4.3.3:
>f  :1.4.3.4:
>g  :1.4.3.5:
>h  :1.4.3.6:
>i  :40:
>j  :41:
>k  :42:
>l  :43:
>m  :44:                                                              1 1
>n  :45:
>o  :46:                                                  1
>p  :47:                                                  1
>q  :48:                                                  1
>r  :49:
>s  :50:
>t  :51:                1
>u  :52:
>v  :53:
>w  :54:
>x  :55:                                                    1 1
>y  :65:                                                  1
>z  :66:
<a  :67:                                                  1
<b  :68:
<c  :70:                                                  1
```

```
                                        72
>a   :1.4.1:
>b   :1.4.2:                            1
>c   :1.4.3.1:
>d   :1.4.3.2:
>e   :1.4.3.3:
>f   :1.4.3.4:
>g   :1.4.3.5:
>h   :1.4.3.6:
>i   :40:
>j   :41:
>k   :42:
>l   :43:
>m   :44:
>n   :45:
>o   :46:
>p   :47:
>q   :48:
>r   :49:
>s   :50:
>t   :51:
>u   :52:
>v   :53:
>w   :54:
>x   :55:
>y   :65:
>z   :66:
<a   :67:
<b   :68:
<c   :70:
```

```
                        202122232425262728293031
>a  :1.4.1:              1
>b  :1.4.2:                 1 1 1 1   1 1 1 1   1
>c  :1.4.3.1:                                 1
>d  :1.4.3.2:
>e  :1.4.3.3:               1                 1
>f  :1.4.3.4:
>g  :1.4.3.5:
>h  :1.4.3.6:               1
>i  :40:                          1 1
>j  :41:                    1
>k  :42:                          1 1 1         1
>l  :43:                    1     1   1
>m  :44:
>n  :45:
>o  :46:
>p  :47:
>q  :48:
>r  :49:
>s  :50:
>t  :51:
>u  :52:
>v  :53:
>w  :54:
>x  :55:
>y  :65:
>z  :66:
<a  :67:
<b  :68:
<c  :70:
```

```
                    3233343536373839404142434445464748495051525354555657585960616263646566
>a  :1.4.1:
>b  :1.4.2:
>c  :1.4.3.1:       | | |
>d  :1.4.3.2:             | | | | |
>e  :1.4.3.3:         |             | |
>f  :1.4.3.4:           |               | |     | |
>g  :1.4.3.5:           |                   |
>h  :1.4.3.6:                   | |             | | |
>i  :40:                        |                                   | | | | | | | | | | |
>j  :41:                                                      |
>k  :42:                                                                      | |
>l  :43:
>m  :44:                                      | | | |
>n  :45:                                              |
>o  :46:                                                |
>p  :47:                        |             |
>q  :48:                        |                                 |
>r  :49:
>s  :50:
>t  :51:
>u  :52:
>v  :53:
>w  :54:
>x  :55:                                      |   |
>y  :65:                                                      |
>z  :66:                                      | |
<a  :67:                                                      |
<b  :68:                                      | |
<c  :70:
```

		67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93
>a	:1.4.1:																											
>b	:1.4.2:																											
>c	:1.4.3.1:																										I	
>d	:1.4.3.2:																										I	
>e	:1.4.3.3:																											
>f	:1.4.3.4:																										I	
>g	:1.4.3.5:																										I	
>h	:1.4.3.6:																										I	
>i	:40:	I	I	I	I		I	I	I																		I	
>j	:41:									I	I	I	I	I	I													
>k	:42:		I	I		I										I												
>l	:43:		I	I			I							I			I	I	I	I	I							
>m	:44:																					I						
>n	:45:																											
>o	:46:																											
>p	:47:																											
>q	:48:																											
>r	:49:																							I				
>s	:50:																											
>t	:51:																											
>u	:52:																											
>v	:53:																											
>w	:54:																											
>x	:55:																											I
>y	:65:																						I					
>z	:66:																											
<a	:67:																								I			
<b	:68:																											
<c	:70:																									I		

1) requires the ability to invoke :3:, ERROR, with a string message, which does not return but handles further processing
2) requires ability to invoke NEXTCARD which makes a new card image available, i.e. ability to invoke :2: and returns to the invoker only if a card for the current algorithm is available
3) NEXTCHAR is the value of the character which immediately follows the character of the current card image produced by the last call of NEXTCHAR
4) an end-of-file condition has occurred
5) requires the ability to invoke the termination of the entire program, i.e. :1.5: ENDOFFILE
6) requires the ability to invoke, ALGINIT (:1.1.1:) the start of processing for a new algorithm
7) CP is an index into C and indexes the last character which was produced as a value from NEXTCHAR. After an execution of NEXTCARD, :2:, CP must equal 0
8) write access required for CP
9) C[1] ... C[80] contains the characters, in order, of the card image which is inputted as a result

of the last execution of GETIMAGE
10) requires read access to C
11) requires ability to invoke GETIMAGE which inputs a card and returns to the caller only if a card was inputted
12) "," in columns 1 and 2 indicate that the program is to terminate and a "," in column 1 only indicates that a new algorithm is to be processed
13) requires ability to print the string argument which is passed as the parameter to ERROR
14) ability to perform printing operations
15) requires ability to invoke :1.4: which processes the remaining data images for this algorithm
16) requires write access to PR
17) pr = **true** means "print the register after each successful application of a rule; otherwise d not print the register after each successful application of a rule
18) MAXA equals the maximum number of alphabets permitted for an algorithm
19) requires read access to MAXA
20) requires ability to set the failure routine for NEXTCHAR, i.e. the ability to invoke SETFAIL with a variable which names the part which is to be invoked if no more characters are available from NEXTCHAR
21) NG contains the number of generic variables encountered for the current algorithm
22) CG[i], $1 \le i \le$ NG, equals the i-th generic variable encountered for the current algorithm
23) AG[i], $1 \le i \le$ NG, equals the alphabet name which CG[i] is a generic variable
24) a failure routine has been set if NEXTCHAR cannot provide additional characters from the current image
25) read access to DOT, COMMA, COLON, SEMI, OPEN, CLOSE which contain the values of ".", ",", ":", ";", "(", ")"
26) assumes write access to CHAR
27) assumes read access to CHAR
28) requires ability to invoke RULES which names the rule input part, but since RULES is a label in the main program a **go to** statement can be used
29) ERRHEAD assumes compete control when invoked and handles error messages and further processing
30) NA equals the number of alphabets which have been processed thus far for the current algorithm
31) requires read accesss to NA
32) requires write access to NA
33) requires read/write access to NOIT, which controls a loop that process alphabets
34) requires ability to invoke STORALPH, which stores the alphabet character if all requirements are met,otherwise STORALPH invokes appropriate error routines
35) requires read/write access to NOIT1, which controls a loop

that processes generics
36) requires ability to invoke STORGEN which stores the content of CHAR, if legal, otherwise invokes the apporpriate error
37) requires ability to invoke ALPHFIN which cmpletes any needed processing after an entire alphabet has been stored
38) either no alphabets have been processed of all alphabets processed have been correct
39) requires ability to invoke FAIL which correctly determines which objects assume control
40) read access required for CP
41) read and write access required for the variable I
42) requires ability to invoke TESTLEGAL which returns only if CHAR is not ".", ",", ";", ":"
43) requires ability to invoke TESTGEN which returns only if CHAR is not equal to an already used generic variable for this algorithm
44) A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters
45) requires read access for A
46) NG contains the number of generic variables encountered for the current algorithm
47) requires read access to NG
48) requires write access to NG
49) requires write access to AG
50) CHAR contains the next unstored character from the alphabet or generics being currently processed
51) AL[1] ... AL[NA] names the index of the lower bound of the characters in an alphabet, i.e. AL[i] is the lower bound for the i-th alphabet and AL[i-1] is the upper bound for that alphabet, where AL[0] equals the initial value plus 1 of UPA
52) requires read access to AL
53) requires ability to invoke ERRGEN, which assumes control and invokes an appropriate error routine
54) requires ability to invoke TESTAL which returns only if CHAR is not equal to a character which has already occurred in the alphabet currently being processed
55) requires ability to invoke TEST which returns only if there storage space as indicated by the values of LPA and UPA
56) UPA names the last cell of an array into which an alphabetic character was stored, counting from the top of some one-dimensional array. UPA is decremented by 1 each time an available cell needs to be named
57) requires read access to UPA
58) requires write access to UPA
59) A names a one-dimensional array, which from UPA to its upper bound contains alphabetic characters
60) requires write access for A

61) requires write access to AL
62) LPA names the las cell of an array into which rules are stored, counting from the lower bound of a one-dimensional array . LPA is incremented by one each time an available cell needs to be named
63) requires read access to LPA
64) LPA and UPA are index variables for the same array
65) requires read access to CG
66) requires write access to CG
67) requires read/write access to J
68) requires read/write access to K
69) requires read access for A
70) requires ability to invoke INITRA, which initializes the input part for rules for a new algorithm
71) requires ability to invoke CARDISRULE which determines whether the current card image is to be interpreted as as a rule or not
72) requires ability to invoke PRINTCARD which prints the current card image
73) requires ability to invoke EDR, end of rules condition processor
74) requires ability to invoke INITR which initializes for inputting a new rule
75) requires ability to invoke LABL which collects a label terminated by the character in TERM and leaves the integer label value in LAB
76) requires ability to invoke STORLABEL which associates the label with the current rule
77) requires write access to TERM
78) ability to invoke STORCHAR which returns only if the content of CHAR could be successfully stored with rule being processed
79) requires ability to invoke INITRIGHT which initializes for processing the input of a right part of a rule
80) requires ability to invoke PROCTERM which indicates a rule as a terminal rule; requires ability to invoke PROCSUC which processes the successor part of a rule
81) requires read access to LAB
82) a negative value in LAB indicates that no label was collected
83) after rule initialization, A[N1] - 1 = NR, N1 names the current rule A[N1] = -1, N2 = N1+1, A[N2] = 2, NE = N1 + 2 and the location which is the name of the immediately preceding rule names the current rule. thus if N1' is the value of N1 prior to this initialization then A[N1'] = N1 - unless NR = 1 in which case the previous value of N1 is not defined
84) R names a one-dimensional array which contains pointers to rules such that label i names rule R[i]; R[i] = 0 means label i is not defined;

R[i] > 0 means the label is defined and
R[i] is a pointer to the rule it namws;
R[i] < 0 means that label i is undefined but has
been referenced by rule -R[i] and is
the head of a chain

85) requires write access to R
86) requires read access to R
87) A names a one-dimensional array which from its
lower bound to LPA contains representations of rules
88) requires read access to NE
89) NE names a cell in A such that A[NE] indicates the current number
of characters in the rule part being processed - left part
if the left part is being processed or right part if the
right part is being processed
90) requires write access to LPA
91) at most 100 labels are permitted and their definitions
appear in the one-dimensional array R[1:100] such that
R[i] contains the definition of label i when the label
is interpreted as a positive integer
92) NR indicates the rulename which is currently
being processed
93) requires read access to NR
94) requires write access to NR
95) requires write access to NE
96) TERM contains the non-digit character which is expected
to terminate a label
97) requires read access to TERM
98) the digits "0", "1", "2", ... , "9" are represented by character
codes such that "0" - ZERO = 0, ... ,"9" - ZERO = 9
and the only legal label characters are digits and BLANK
99) read/write access to LAB is required
100) legal range of labels is 1 through 100
101) BLANK contains the representation of a space and ZERO
contains the representation of a zero
102) requires read access to BLANK and ZERO
104) assumes LAB contains a legal label name or -1 which
indicates that no label has been concatenated
105) requires write access to LAB
106) N1 names the rule being currently inputted
107) requires read access to N1
108) ERRULENAME contains the value which indicates a routine
which can take control if an error is discovered as rules
are being stored
109) requires read access to ERRULENAME
110) requires read access to N2
111) requires read/write access to L
112) requires write access to N2
113) a colon in column 4 when processing rules indicates that
the card is to be interpreted as a rule
114) requires read/write access to Q

115) a value of -1 in A[N2] indicates that the
rule is a terminal rule
116) ERRHEADNAME is a variable which names the routine ERRHEAD
117) requires read access to ERRHEADNAME
118) RULESNAME is a variable which names the routine RULES which
inputs the rules for an algorithm
119) requires read access to RULESNAME
120) assumes all rules have been inputted correctly;
assumes that the current card isrepresents an initial register
contents; assumes that NEXTCARD returns control only if
a data image is available for the current algorithm
121) requires the ability to invoke :4:, ERRORE, with a string
message, which does not return but handles further processing
at the execution stage
122) ERRDATANAME contains a value which indicates a routine
which indicates a routine which can take control if an error is
discovered while an initial register contents is being input
123) requires read access to ERRDATANAME
124) REG, is a one-dimensional array which contains
the characters in the register
125) assumes read access to REG
126) assumes write access to REG
127) requires read/write access to RPP which is used as
a temporary register position pointer when the register
is being initially filled
128) requires read/write access to RC which is used
to contain single characters from the current data card when
the register is being filled
129) assumes read access to MAXRL which contains the maximum number
of characters permitted in the register
130) requires write access to MAXT1
131) requires write access to RL
132) the name of the first rule is 2
133) MAXT1 contains the number of trial rule applications
still permitted for this execution of an algorithm
134) write access to RULENAME required
135) requires read access to TRM
136) FIN is a result of NOSUC which indicates that there are no more
rules which can be applied
137) requires read access to FIN
138) read access to RULENAME required
139) TRM is a result of SUCSUC and UNSUCSUC which
indicates that the algorithm should terminate, i.e. TERM
names no legal and indicates termination
140) requires ability to invoke SEARCH which
searches for a match of the left part of rule
RULENAME, and which returns the value **true** if
a match is found, **false** otherwise
141) requires read access to MAXT1
142) requires ability to invoke REPLACE, which has the effect of

replacing the register contents with the right part of the
rule named by RULENAME, where the left part was matched

143) requires the ability to invoke SUCSUC which has the value of the success to ruie RULENAME for a successful application of rule RULENAME

144) requires the ability to invoke UNSUCSUC which has the value of the name of the successof to rule RULENAME after an unsuccessful application of rule RULENAME

145) assumes A[RULENAME + 1] names the successor for rule RULENAME for a successful application of rule RULENAME

146) requires read a cess to PR

147) requires the ability to invoke PRINTREG which prints the contents of the register

148) if RULENAME = TRM then dot termination has occurred, otherwise the rules have been exhausted

149) G is a one-dimensional array such that G[i] = -1 if generic variable i is not defined after a successful search for a left part of a rulei, otherwise G[i] > 0
and is the character corresponding to the generic i

150) requires read access to G

151) requires write access to G

152) requires read/write access to NMAT

153) assumes that a generic in a rule has been saved as a negative value

154) assumes Q is a legal character pointer into the left
part of a rule whose first character is P + 3

155) assumes A[RULENAME] names the successor rule for rule RULENAME after an unsuccessful application of rule RULENAME

156) requires the ability to invoke LEFT which names the location which is the first character of the left part of rule, RULENAME

157) requires ability to invoke LENGTHL which has the value of the number of characters in the left part of the rule named by its parameter

158) LEN names the number of the character in the left part of the rule currently being processed

159) requires read access to LEN

160) requires write access to LEN

161) LFT names the starting location for the left part of the current rule name

162) requires read access to LFT

163) requires write access to LFT

164) requires read/write access to NOSUC

165) RL contains the current register length

166) requires read access to RL

167) requires ability to invoke RULLFTCHR(A,B) which produces the value of the B-th character of the left part which starts at A

168) RP names the character position where a character sequence is to be replaced

169) requires read access to RP

170) requires write access to RP
171) assumes L1 $\neq$ L2
172) requires ability to invoke GENERIC which has the value **true** if its parameter represents a generic variable
173) requires read/write access to CC
174) requires read/write access to LHP
175) requires ability to invoke ADJUST whose first parameter indicates the number of characters in the left part of a rule and whose second parameter indicates the number of characters in the right part of the rule. ADJUST modifies the register, if necessary, so that the right part can be inserted where the matched left part is
176) requires ability to invoke INSERT which inserts the appropriate right part over the matched left part
177) requires ability to invoke LENGTHR which returns the value of the number of characters in the right part of the rule named by its parameter
178) requires read/write access to LEN and LENR
179) requires read access to LENR
180) requires write access to LENR
181) requires read access to L1, L2, and TP; L1 contains the length of a character sequence being replaced by a character sequence of length L2
182) requires read write access to TQ and CQ
183) requires ability to invoke RULRTCHAR(A) which returns the character which is the A-th character of the right side of rule RULENAME
184) requires the ability to invoke UNDEF(A) which returns the value **true** if A is an undefined generic variable representation for this rule application; **false** otherwise
185) requires ability to invoke VALG(A) which produces the character associated with the generic representation A for this rule application
186) assumes LENR contains the length of the string being inserted
187) requires read access to AG
188) assumes Y is a legal character pointer into the right part of a rule whose first character is named by X + LENGTHL(X) + 4
189) assumes X + 2 is an index which names the first character of the left part of a rule -1
190) assumes A[X + LENGTHL(X) + 3] is the number of characters in the right part of rule X
191) assumes read access to X and assumes that the length of the left part of a rule named by X is contained in A[X + 2]
192) requires read access to RULENAME
193) reauires read/write access to QK

An analysis of the final stage using the measure has led to the following good decomposition

```
((a) ((>n) ((>r) ((>t) ((r) ((>a) ((e) ((f)
(( >s, >u, >w) ((s) ((*x) ((i) ((k, o) ((i) ((g, h)
((v, w) ((x) ((*a, *b, *c, *d, *f, *g, *h, *i, *j, *l)
((>c, >d, >f, >g) ((<b, >z, >x, >m)
((>o, >y, >p, >q, <c, <a) ((>e, >h, >v)
((*s, *r, *q, *u, *v) ((u, y) ((>j, >k, >i, >l, >b)
((z, t) ((*o, *p, *w, *t) ((m, p, q) ((*k, *n, *m))
1.56 ) 1.65 ) 1.52 ) 1.60 ) 1.35 ) 1.41 ) 1.20) .92 )
.52 ) .82 ) 1.08 ) 1.25 ) 1.02 ) .65 ) .77 ) .32 ) .34 )
.53 ) .50 ) .47 ) .22 ) .04 ) .04 ) .04 ) .04 ) .05 ) .05 ) 0
```

This decomposition is not the best decomposition, but it illustrates the result of using the measure to produce a good decomposition which satisfies several of the properties stated in the Introduction. Specifically, those objects which manipulate the representation of rules appear together. This situation along with others demonstrates that the decomposition appears to have several of the independence properties stressed by Parnas. The parts FAIL, SETFAIL, and the initialization portion, though not elaborated here, localize detailed information about the flow of control in the program and interact little with the previous elaborations.

APPENDIX II: COMMENTS ON A NOTE ON COMPILER STRUCTURE

McKeeman[MK1], in a paper entitled "Compiler Structure", has presented several guides that aid in fragmenting a compiler into modules. Does such a modularization possess good structure in the sense of the definition and measure presented in this thesis? It is probably the case that these modularizations could have good structure, but good structure is not guaranteed.

The two kinds of fragmentation discussed are **vertical fragmentation** and horizontal **fragmentation**.

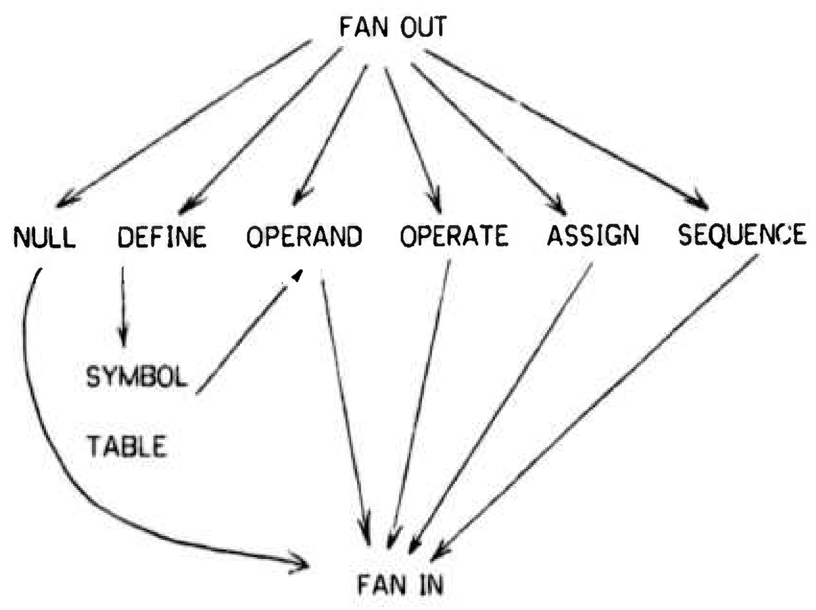
Vertical fragmentation corresponds to decompositions whose modules may be regarded as "passes" or "phases" in the compilation process. Each module accepts as input the output of a previous module. Thus, a possible vertical decomposition for a compiler is

$$\begin{tikzcd}
\text{INPUT} \arrow[d, Rightarrow] & \text{(analyze text to produce a sequence of single characters)} \\
\text{SCAN} \arrow[d] & \text{(analyze single characters to produce tokens)} \\
\text{PARSE} \arrow[d] & \text{(analyze tokens to produce a parse tree)} \\
\text{SYNTHESIS} \arrow[d] & \text{(analyze parse tree to produce a computation tree)} \\
\text{GENERATE} \arrow[d] & \text{(analyze computation tree to produce language specific code)} \\
\text{EMIT} \arrow[d] & \text{(analyze language specific code to get machine specific code)} \\
\text{OUTPUT} & \text{(analyze code to get executable or loadable text)}
\end{tikzcd}$$

McKeeman stresses the importance of precisely describing the intermediate languages that connect the modules and provides examples of such intermediate languages. Hence, pairs of successive modules share information about their intermediate languages. This can lead to many interactions between adjacent modules. An alternative that has been displayed in Appendix I and in the work of Parnas, is to provide additional modules which make information about the intermediate languages available. This eliminates the need for sharing the entire grammar of the intermediate language. Thus changes to these intermediate languages correspond to adding or deleting or changing functions at the interface. The measure indicates that the modules in this alternative interact less than in the modularization suggested by McKeeman - at the expence of requiring additional modules.

horizontal fragmentation can be used to further fragment modules in some **vertical fragmentation**.

McKeeman presents the following **Horizontal fragmentation** for the module SYNTHESIS



FAN OUT is intended to pass appropriate parts of the phrase structure tree or canonical parse to the modules beneath it. FAN IN, recombines the results of these modules into a computation tree or sequence of actions.

The final decomposition shown in Appendix I and the modularizations suggested by Parnas[PA1-5] indicate that the boundaries of these modules may not be as "clean" as the diagram suggests. For example, information contained in the SYMBOL TABLE module may be required by more modules than are indicated by the arrows. (Indeed, similar kinds of modules displayed in [MK] share far more assumptions than are indicated by either diagrams or text.) Further, assumptions made by different modules may suggest additional modules in order to maintain the independence properties suggested by the diagram. Indeed, FAN IN and FAN OUT may well share enough assumptions to warrant large parts of them to be written as additional modules.

Stronger statements than these can only be made if more detailed information about the intended behavior of the modules is presented.